State-of-the-art in String Similarity Search and Join

Sebastian Wandelt
Knowledge Management in Bioinformatics, HU Berlin, Berlin, Germany

Dong Deng
Tsinghua University, Beijing, China

Stefan Gerdjikov
FMI Sofia University, Sofia, Bulgaria

Shashwat Mishra
Special Interest Group in Data, IIT Kanpur, Kanpur, India

Petar Mitankin
IICT Bulgarian Academy of Sciences, FMI Sofia University, Sofia, Bulgaria

Manish Patil
Louisiana State University, Louisiana, USA

Enrico Siragusa
Algorithmic Bioinformatics, FU Berlin, Berlin, Germany

Alexander Tiskin
Department of Computer Science, University of Warwick, United Kingdom

Wei Wang
University of New South Wales, New South Wales, Australia

Jiaying Wang
Northeastern University Shenyang, China

Ulf Leser
Knowledge Management in Bioinformatics, HU Berlin, Berlin, Germany

ABSTRACT

String similarity search and its variants are fundamental problems with many applications in areas such as data integration, data quality, computational linguistics, or bioinformatics. A plethora of methods have been developed over the last decades. Obtaining an overview of the state-of-the-art in this field is difficult, as results are published in various domains without much cross-talk, papers use different data sets and often study subtle variations of the core problems, and the sheer number of proposed methods exceeds the capacity of a single research group. In this paper, we report on the results of the probably largest benchmark ever performed in this field. To overcome the resource bottleneck, we organized the benchmark as an international competition, a workshop at EDBT/ICDT 2013. Various teams from different fields and from all over the world developed or tuned programs for two crisply defined problems. All algorithms were evaluated by an external group on two machines. Altogether, we compared 14 different programs on two string matching problems ($k$-approximate search and $k$-approximate join) using data sets of increasing sizes and with different characteristics from two different domains. We compare programs primarily by wall clock time, but also provide results on memory usage, indexing time, batch query effects and scalability in terms of CPU cores. Results were averaged over several runs and confirmed on a second, different hardware platform. A particularly interesting observation is that disciplines can and should learn more from each other, with the three best teams rooting in computational linguistics, databases, and bioinformatics, respectively.

Keywords

String search, String join, Scalability, Comparison

1. INTRODUCTION

Approximate search and join operations over large collections of strings are fundamental problems with many applications. String similarity search is used, for instance, to identify entities in natural language texts [29], to align DNA sequences produced in modern DNA sequencing with substrings of a reference genome [16, 17], or to perform pattern matching in time series represented as sequences of symbols [10]. String similarity joins are building blocks in the detection of duplicate Web pages [13], in collaborative filtering [2], or in entity reconciliation [7]. Research in this field dates back to the early days of computer science and the area is still highly active today. Literally hundreds of methods have been proposed.
For string similarity search and join, fundamental techniques include seed-and-extend methods (turning similarity search into an exact search problem of smaller strings, e.g. All-Pairs [2], ED-Join [31], and PPJoin [32]), partitioning techniques (e.g. Pass-Join [15], NGPP [29], and PartEnum [1]), prefix-filtering methods (e.g. Trie-Join [8] and PEARL [23]), and other methods (e.g. M-Tree [12], LSH [9], and FastSS [25]).

Research in the field has been carried out in various scientific disciplines, the most important ones probably being algorithms for pattern matching, computational linguistics, bioinformatics, and database/data integration. There are subtle differences between the problems being studied, for instance varying in the concrete similarity measure (edit distance, Jaccard, Hamming etc.), the type of string comparisons (global or local alignment, approximate substring search etc.), the amount of indexing being allowed (online in the queries and/or the database). Methods often are tuned for specific ranges of allowed error thresholds or query lengths, specific hardware properties, specific alphabet sizes, or specific distributions of errors. Though newly published methods mostly compare to some prior works, selection of these works is often suboptimal and comparisons are carried out on different data sets; data sets all too often are not made publicly available, which means that results are not reproducible. In Figure 1, we show existing evaluation results for the most relevant work on string similarity search/join with edit distance constraints. As a consequence of the heterogeneity of approaches and problems, the lack of common benchmarks, and the dispersal of research in different communities, today it is hardly possible to choose the best algorithm for a given problem.

In this work, we report on the (to the best of our knowledge) most comprehensive benchmark in two specific string similarity match problems to date: $k$-approximate search and $k$-approximate join (with $k$ as an edit distance threshold; see below for exact definition). We organized this benchmark using a rather uncommon approach: The *International competition on Scalable String Similarity Search and Join (S4)*\(^1\) held as a workshop in conjunction with EDBT/ICDT 2013. We made an open, worldwide call for contributions and provided

---

\(^1\)http://www2.informatik.hu-berlin.de/~wandelt/searchjoincompetition2013/
tion 2. All submitted methods are briefly presented in Section 3. Evaluation results for approximate string searching are presented in Section 4 and for approximate string join in Section 5. In Section 6, we discuss the results of the competition and compare results to three external programs, Flamingo [3], Pearl [23], and SSI [9], which were evaluated after our competition was finished. The paper is concluded with Section 7.

2. BACKGROUND

We define the problems of approximate string searching and approximate string join. Our competition and evaluation methodology is introduced together with a description of datasets and evaluation environments.

2.1 Formal problem statement

**Definition 1 (Strings).** A string $s$ is a finite sequence of symbols over an alphabet $\Sigma$. The length of a string $s$ is denoted by $|s|$ and the substring starting at position $i$ with length $n$ is denoted by $s(i,n)$. We write $s(i)$ as an abbreviation for $s(i,1)$. All positions in a sequence are zero-based, i.e., the first character of $s$ is $s(0)$.

As a distance function between two strings we use unweighted edit distance for different error thresholds $k$.

**Definition 2 (String similarity).** Given strings $s$ and $t$, $s$ is $k$-approximately similar to $t$, denoted $s \sim_k t$, if and only if $s$ can be transformed into $t$ by at most $k$ edit operations. The edit operations are: replacing one symbol in $s$, deleting one symbol from $s$, and inserting one symbol into $s$.

We investigate two problems: string similarity search and string similarity join.

**Definition 3 (Similarity search).** Given a collection of strings $S = \{s_1, \ldots, s_n\}$, a query string $q$, and an edit distance threshold $k$, the result of string similarity search of $q$ in $S$ is defined as

$$\text{SEARCH}(S, q, k) = \{s_i \in S \mid s_i \sim_k q\}.$$  

For instance, given a collection $S = \{ACA, TGA, AC\}$, a query string $q = ACA$, and $k = 1$, the result of string similarity search is $\text{SEARCH}(S, q, k) = \{1, 3\}$.

**Definition 4 (Similarity (self) join).** Given a collection of strings $S = \{s_1, \ldots, s_n\}$ and an edit distance threshold $k$, the result of string similarity self-join of $S$ is defined as $\text{JOIN}(S, k) = \{(i,j) \mid s_i \in S \land s_j \in S \land s_i \sim_k s_j\}$.

For instance, the result of a string similarity self-join on data set $S$ from above with $k = 1$ is $\text{JOIN}(S, 1) = \{(1,1), (1,3), (2,2), (3,1), (3,3)\}$. Note that we explicitly include the reflexive and symmetric closure in our definition. We note that a self-join is comparable to a join between two different sets as we make no assumptions about the a priori average level of similarity of the strings in a set. In the following we will often use the term join instead of self-join.

2.2 Competition and methodology

This competition brought together researchers and practitioners from database research, natural language processing, and bioinformatics. The challenge for all participants was to perform string similarity search and join over unseen data and query sets with varying error thresholds $k$ as fast as possible. The call for the competition was circulated by email through various lists addressing the different areas dealing with string matching, in particular databases, algorithms, computational linguistics, and bioinformatics. We also contacted directly a few dozen researchers known for their contributions to the field. The different phases of the competition are shown in Figure 2.

In total we received initial expressions of interest from 22 teams, out of which 11 teams officially submitted a program. One team failed to hand in a complete paper describing their approach on time, and another group withdrew shortly before the final deadline. Thus, we eventually compared programs from 9 teams (see Table 1). All these teams gathered at a workshop co-locates with EDBT/ICDT 2014 in Genoa, where each team presented its approach and the results of our evaluation were discussed. This format led to a workshop in the best sense of the word - as all presentations essentially covered the same problems, talks were highly focused and intensive discussions and exchanges of ideas emerged naturally. We also organized culinary prices for the best teams which were immediately shared with the entire audience.

We succeeded in reaching out to different research communities: two teams have their home in bioinformatics, two in computational linguistics, one in algorithms/computational complexity, and the remaining four are best described as database groups. Contributions came from four continents and seven countries. At least six teams (Team 1, 3–5, 7–9) published highly influential papers on string matching problems before [15, 19, 22, 24, 26, 28, 33], while two teams (Team 2 and Team 6) can be considered as newcomers. As Table 1 shows, the techniques used cover a broad range and thus subsume a large fraction of previous research in $k$-approximate string matching. In addition, out of the five non-dominated methods in Figure 1, four methods are directly represented by corresponding authors in our competition.

The competition consisted of two tracks:

**Track 1:** Given a set of strings $S$, a query string $q$ and an error threshold $k$, compute $\text{SEARCH}(S, q, k)$.

**Track 2:** Given a set of strings $S$ and an error threshold $k$, compute $\text{JOIN}(S, k)$.

Small subsets of the final evaluation datasets (around 5%) were made available for the contestants for preparation of their submissions. It was announced that these
strings are representative for the whole evaluation datasets. Furthermore, we announced a description of the evaluation hardware and provided a virtual machine mirroring the software environment used for evaluation. Thus, all teams could develop and tune their programs before submission. Each program was allowed to use any number of threads, with the restriction that the official evaluation environment System 1 (see below) has 8 cores, and a maximum of 48 GB of main memory. Details on CPU, clock rate, cache sizes, disks etc. were not provided to prevent hardware specific tuning; note that this implies that further improvements could be possible taking the specific hardware into account [20]. Programs were allowed to have two phases, one for indexing the data set, and one for evaluating a set of queries on the set (or the index). The main evaluation criterion was measured wall clock time. In general, we ranked programs based on average runtime over three independent runs; variations in runtime were very low and are not reported here. If programs ran much longer than most of the competitors, experiments were only performed once. We also measured the indexing time and report it here, but we did not take it into account for ranking.

2.3 Datasets

We used two different types of datasets, for evaluation in both tracks, to cover different alphabets and string lengths. Each type of dataset contains five distinct, highly-similar datasets of increasing size, for evaluating scalability.

**READS:** These data sets contain reads obtained from a human genome. The data is characterized by a small alphabet (5 symbols) and quite uniform length of strings (around 100 symbols per string).

**CITIES:** These data sets are based on geographical names taken from World Gazetteer. The data is characterized by a larger alphabet (around 200 symbols) and non-uniform length of strings (5-64).

Considered values for k depend on the dataset. For READS, we announced and used k ∈ {0, 4, 8, 12, 16}; for CITIES k ∈ {0, 1, 2, 3, 4}. Thus the maximum error rate for READS is around 1/5 and for CITIES around 1/3. The size of each dataset and the number of queries for Track 1 are shown in Figure 3. For READS, the number of reads starts with 15,000 (TINY) and ends with 150,000,000 (HUGE). For CITIES, the number of cities starts with 10,000 (TINY) and ends with 1,000,000 (HUGE). For READS and CITIES, the maximum number of queries in HUGE is 100,000.

2.4 Evaluation Environments

After the development phase of the competition, participants submitted their final programs which were evaluated on two different platforms.

**System 1:** A computer with 8 cores (processor: AMD FX-8320) and 64 GB RAM. The operating system (Fedora Scientific 17 x86_64) was installed on a SSD with 128 GB. The SSD contained the datasets as well as the programs. Each program serialized its results to an external USB 3.0 hard disk with 3 TB. This system was announced beforehand and results for this system were used for ranking.

**System 2:** A server with 80 cores (processors: Intel Xeon CPU E7 - 4870) and 1 TB RAM. The operating system was openSUSE 12.1 x86_64. All datasets, programs, and serialized results were put
3. METHODS

This section describes the methods used by each team in their submissions to the competition.

3.1 Team 1

PassJoin (Tsinghua University) adopts a partition-based framework for string similarity search and joins. The basic idea is that given two datasets $R$ and $S$, and an edit distance threshold $k$, each string in $R$ is split into $k + 1$ disjoint segments. For each string in $S$, PassJoin checks if it contains any substring matching the segments of $R$. If no, PassJoin prunes the string; otherwise the string and those strings whose segments match the substrings of the string are verified. There are two challenges in the partition-based method. The first one is how to select the substrings. A position-aware substring selection method and a multi-match-aware substring selection method have been proposed. It has been proven the multi-match-aware substring selection method selects the minimum number of substrings. And it is the only way to select the minimum number of substrings when the string length is longer than $2 \times k + 1$. The second one is how to verify each candidate pair. PassJoin uses a length-based verification method, an improved early termination technique, and an extension-based verification method.

Team 1 submitted two programs: Program 1_A and Program 1_B. Both programs of Team 1 were evaluated for both tracks and both datasets.

3.2 Team 2

Team 2 (Magdeburg University) tries to outperform conventional index-searches by a sequential search algorithm. Starting from a naive algorithm for computing edit distances, several optimizations are introduced. Calculation of the edit distance is improved by using length-heuristics. If the computation of a dot matrix cannot be avoided, the program applies several heuristics to prune the search space early. Further optimizations include the use of reference-based semantics over value-based semantics and the use of simple data types. They devise simple scheduling strategies depending on the current workload.

Team 2 submitted only one program: Program 2_A, which was evaluated for Track 1 only.

3.3 Team 3

The Waterfall algorithm of Team 3 (University of Warwick) solves the competition challenge without indexing or any other preprocessing of the database strings. First, a reduction of the edit distance problem to the longest common subsequence (LCS) problem between the database string and the query string, both suitably modified, is applied. The strings’ LCS score is then computed by a bit-parallel algorithm, based on [6]. This technique is extended so that a database string can be tested simultaneously against multiple query strings, by a subword-parallel technique similar to that of [14], which was further developed in the waterfall algorithm. Due to the self-imposed restriction of not preprocessing the database, the algorithm runs significantly slower than other competitors, which do index the database strings before answering the queries. However, the approach chosen by Team 3 can prove useful in a situation where input preprocessing is not possible. Such a situation occurs e.g. when the string database is replaced by a continuous stream of input strings, each of which needs to be matched against a small set of query strings in real time.

Team 3 submitted only one program: Program 3_A, which was evaluated for both tracks and both datasets.

3.4 Team 4

The WallBreaker of Team 4 (Sofia University) is a new sequential algorithm for the similarity search problem in a finite set of words. It reduces and essentially overcomes the wall-effect caused by the redundantly generated false candidates. To achieve this the query is split into smaller subqueries with smaller threshold. This allows to start with an exact match and then extend these exact matches to longer candidates whereas the threshold increases slowly in a stepwise manner. In order to implement this idea in practice two kind of resources are used: (i) a linear space representation of the infixes in the finite set of words that enables a left/right extension of an infix in constant time per character; and (ii) efficient filters, universal Levenshtein automata [18], synchronised Levenshtein automata [19] and standard Ukkonen filter [27], that prune the unsuccessful candidates as soon as a clear evidence for this occurs. In the index structure information about the possible lengths of longest/shortest left/right possible extensions are encoded. This information is then used as an additional length-filter.

As a result a breaking-the-wall-effect is achieved. In the beginning the WallBreaker considers only small neighborhoods of short words which keeps the searching space modest. Afterwards, while increasing the potential size of the neighborhoods, longer infixes are generated that are much more informative than shorter ones and suppress the searching space for their own sake. For further details the reader is refered to [11], where besides the standard Levenshtein edit-distance also the generalized Levenshtein edit-distance is handled.

Team 4 submitted two programs:

Program 4_A: It uses 16 threads, the additional length-filter, and applies universal Levenshtein automata for
thresholds $\leq 5$, and synchronised Levenshtein automata for thresholds $\leq 3$.

**Program 4.B:** It uses 16 threads, ignores the additional length-filter, and applies universal Levenshtein automata for thresholds $\leq 5$, and synchronised Levenshtein automata for thresholds $\leq 3$.

Both programs of Team 4 were evaluated for both tracks and both datasets.

### 3.5 Team 5

The methods of Team 5 (FU Berlin) are variations of those applied in Masai [24], a tool for mapping high-throughput DNA sequencing data. First an online solution for computing edit distances using a banded version of the Myers bit-vector algorithm [21] is proposed. Team 5 is able to check in time $O\left(k(\frac{(k+1)m+n}{w})\right)$, where $w$ is the CPU word size and $\Sigma$ the string alphabet, if two strings of length $m$ and $n$ (w.l.o.g. $m < n$) are within edit distance $k$. Then they propose to index multiple queries in a radix tree and backtrack them into the radix (or suffix) tree of the database. In practice, radix (and suffix) trees are replaced by simpler radix (and suffix) arrays. Multiple backtracking is parallelized with static load balancing and work queues. Finally, as proposed by Navarro and Baeza-Yates [21], a filtering method partitioning queries into approximate seeds is implemented. Such a filtering method combines the previous two methods and works well up to moderate error rates. The programs are implemented in C++ and OpenMP using the SeqAn library.

Team 5 submitted four programs:

**Program 5.A:** An online algorithm.

**Program 5.B:** Partitioning with minimum seed length (10 for READS, 4 for CITIES)

**Program 5.C:** Partitioning with minimum seed length (13 for READS, 5 for CITIES)

**Program 5.D:** Partitioning with minimum seed length (15 for READS, 6 for CITIES)

### 3.6 Team 6

The submission of Team 6 (IIT Kanpur) uses deletion neighborhoods [25]. A $k$-neighborhood is generated for every string $s \in S$. Every string in the $k$-neighborhood is referred to as a key. The underlying index structure is a hash-table which maintains an inverted index on the keys. In order to circumvent the large space requirements, the program only indexes an $L_s$-length suffix for each key. Given a query string $q$ and an edit-distance threshold $k$, first the $k$-neighborhood of $q$, $N_q$, is generated. The list corresponding to every key in $N_q$ is obtained from the index structure. A union of these lists is guaranteed to be a superset of the answer set $\text{SEARCH}(S,q,k)$. For each string $s$ in the generated candidate list, the program uses a length-threshold aware distance computation to verify $s$. In a multi-core environment, the program partitions the entire workload into $k$ equal parts and each part is handled by a single, dedicated thread. Team 6’s idea is that deletion neighborhoods offer a powerful, selective signature scheme to process edit distance queries. Team 6 only participated in Track 1 of the competition. Further, since deletion neighborhoods are only suited for scenarios with larger alphabet size, Team 6’s submission **Program 6.A** was only evaluated on CITIES dataset.

### 3.7 Team 7

The index structure of Team 7 (Louisiana State University) consists of a generalized suffix tree (GST) and a two-level wavelet tree (WT) on its leaves. The first level WT maintains an array of starting positions of all suffixes of GST. For each leaf of this WT, another WT for the difference between the starting position of the suffix and the string length to which it belongs to is maintained. Given $\tau$, $r$, Team 7 obtains $\tau + k$ disjoint partitions of $r$ aiming to balance selectivity of count filtering and frequency of partitioned segments. Then GST and WT are used to obtain inverted list of each partition pre-filtered by “Position Restricted Alignment” that combines the well-know length and position filters. All inverted lists are then merged to retrieve the strings similar to $r$.

Team 7 submitted only one program: **Program 7.A**, which was evaluated for Track 1 with READS only.

### 3.8 Team 8

Team 8 (University of NSW) presents a solution based on tries, which have the advantages of small indexing space, freeness of verification, and computation sharing among strings with common prefixes. The method proposed is a simple adaptation of trie-based error-tolerant prefix matching [30]. Existing trie-based methods process a query by incrementally traversing the trie and maintaining a set of trie nodes (called active nodes) for each prefix of the query. One common drawback is that they have to maintain a large number of active nodes. Instead, Team 8 records only a small number of potentially feasible nodes as “active nodes” during query processing, which reduces the overhead of maintaining nodes and reporting results. In addition, Team 8 characterizes the essence of edit distance computation by a novel data structure named edit vector automaton, which substantially accelerates the state transition of active nodes, and therefore, improves the total query performance. Naive parallelization is added to exploit multi-core CPUs.

Team 8 submitted only one program: **Program 8.A**, which was evaluated for Track 1 with CITIES only.

### 3.9 Team 9

BWTSearcher of Team 9 (Northeastern University) takes advantage of a cache-aware multicore framework using Burrows-Wheeler-Transform [4]. BWTSearcher segments the whole collection of database sequences to fit to the CPU cache lines. The approximate string search algorithm is based on a partition approach. The query is decomposed into $\tau + 1$ chunks. If $P$ matches the text with at most $\tau$ errors, at least one of the parts will match a substring of the text exactly. A new data structure called BWTPA is proposed to find the matching candidates. Length filter and position filter are used to prune the candidates. Team 9 proposed a reversed segment trie to merge the identical segments, which can save much duplicated computation. In addition, a look
ahead algorithm is developed to support bounded edit distance and improve the verification of the candidate strings. BWTSearcher can search on any dataset, but is not optimized on DNA data, yet.

Team 9 has only one participating program: Program 9_A, which was evaluated on all datasets for Track 1.

4. EVALUATING APPROXIMATE STRING SEARCH METHODS

In the following section we report results for all submissions for Track 1: approximate string search. We present results for READS datasets first and then for CITIES.

4.1 Similarity Search for READS

In Figure 4, we show the indexing and search times for the READS dataset and random values for k (for each query in the dataset we have assigned a random number out of \(\{0, 4, 8, 12, 16\}\)). For READS-TINY and READS-SMALL, most of the programs compute the results within a few seconds, with two exceptions. 2_A, the index-less approach, needs already 185 seconds for answering READS-SMALL. For READS-MEDIUM, 2_A did not compute a result within several hours, so it was not evaluated on the larger datasets. Program 5_A, another index-less approach, needs 23.9 seconds for READS-SMALL and around 45 minutes for READS-MEDIUM. Therefore, 5_A was not tested on READS-LARGE and READS-HUGE.

The fastest programs for READS-HUGE are 4_A and 4_B, taking 232.5 and 249.0 seconds, respectively. The third program is 1_A, which needs 312.1 seconds. However, the indexing time of 1_A is around 20 times shorter than the indexing time for 4_A and 4_B. Programs 1_B, 5_B, 5_C, and 5_D need 10 to 15 minutes for READS-HUGE. Program 3_A, which does not use an index structure, already needs 8 hours to compute all solutions for READS-HUGE.

In Figure 16, we show search times for different values of k and the dataset READS-MEDIUM. The indexing time for all the programs is independent of the value of k, and is shown in Figure 4. Except 3_A and 9_A, all programs can compute the results set for \(k \leq 8\) within a few seconds. The best program for \(k = 16\) is 4_A, needing only 17.8 seconds, followed by 4_B and 1_A. For all values of k, 4_A is among the fastest programs, only clearly outperformed by 1_A for \(k = 12\).

We have further analyzed the effect of batch-processing for all programs for READS-MEDIUM and \(k = 4\), except 2_A. In Figure 15, the average time per query for different numbers of queries is shown. It can be seen that for most programs, the average query answering time per query is reduced, if the number of queries is increased. For a large number of queries, the programs of Team 1 and Team 4 have the shortest time per query.

We have further evaluated the three top-performing programs on our second evaluation environment System 2 with a different number of threads. Each program was preset to use 8, 24, and 80 threads, respectively. In Figure 5, the results of the evaluation are shown. It can be seen that 1_A and 4_A scale quite well with the number of threads: if the number of threads is increased by 3 (8 to 24), the search time is reduced by a factor larger than 2. The improvement from 24 threads to 80 threads is not as big any more. For 5_B there is almost no effect when increasing the number of threads. Their multiple backtracking algorithm is not straightforward to parallelize and the static load-balancing approach doesn’t scale well. In this scenario it is probably easier to abandon multiple backtracking and go back to “standard” single backtracking, to allow a query-by-query parallelization.

4.2 Similarity Search for CITIES

In Figure 6, we show the indexing and search times for the CITIES dataset and random values for k. For
CITIES-TINY and CITIES-SMALL most of the programs compute the results within a few seconds. The only exception are the programs of Team 5, which need already 13.6-39.0 seconds for CITIES-SMALL. All programs were tested on all datasets, with two exceptions. Programs 2_A and 5_A did not return a result for CITIES-HUGE within several hours. Indexing times are quite short for all programs, except 6_A, which almost spends 20 minutes on indexing CITIES-HUGE.

The fastest program for CITIES-HUGE is 1_B, needing 46.8 seconds. It is closely followed by 1_A, 4_A, and 4_B. The programs of Team 5 are the slowest for CITIES, which probably means that their approach is better suited to deal with small-alphabets. In Figure 17, the search times for CITIES-MEDIUM and different values of k are shown. Programs 1_A and 1_B are always among the fastest.

We have further evaluated the three top-performing programs on our second evaluation environment System 2 with a different number of threads. In Figure 7, the results are shown. The results are very similar to the results of READS: Program 1_A and 4_A scale well from 8 to 24 threads and quite good for 24 threads to 80 threads. Program 5_B does not scale as well as the other two (and was not tested for CITIES-LARGE and CITIES-HUGE).

Figure 8 shows a comparison of indexing times vs. search times for READS-HUGE and CITIES-HUGE for System 1.

5. EVALUATING APPROXIMATE STRING JOIN METHODS

In the following section we report on results of all submissions for Track 2: approximate string join. Again, we present results for READS datasets first and then for CITIES.

5.1 Similarity Join for READS

In Figure 18 and Figure 20, we show the join times for the READS dataset, for \( k = 0 \) (a) and \( k = 16 \) (b), respectively.

For \( k = 0 \), all programs have been tested for all datasets, except from 5_A. Program 5_A already needs around 30 minutes to perform a join on READS-SMALL. The fastest programs need less than 10 seconds to perform a self-join on READS-HUGE: 1_A and 1_B. For \( k = 16 \), most programs could only be tested until READS-SMALL. Two programs were evaluated in READS-HUGE: Program 1_A needed 22.9 hours and Program 4_A needed 41.5 hours.

We report the join times for READS-HUGE and different values for \( k \) in Figure 9. Programs 3_A and 9_A already need more than 20 hours to perform a 4-approximate self-join on READS-HUGE. The best performing method is implemented in Program 1_A.

We have further evaluated the three top-performing programs on our second evaluation environment System 2 with a different number of threads. In Figure 21, the results are shown. For all programs a higher number of threads reduces the runtime. It is interesting to see that with an increasing value of \( k \), the effect is bigger than with small numbers. We conjecture that the overhead of setting up the threads and synchronization is dominating for smaller \( k \).

5.2 Similarity Join for CITIES

Join times for the CITIES dataset are reported in Figure 22 for \( k = 0 \) and in Figure 19 for \( k = 4 \). Apart from Program 5_A, all programs finished to compute an exact self-join on all CITIES datasets. Program 1_A is the fastest program in each case. Team 4's programs are ranked second. Program 3_A finishes third, which is quite remarkable for an index-less approach.

The join times for CITIES-HUGE and different values of \( k \) are reported in Figure 10. Program 1_A is the best for all values of \( k \), except for \( k = 1 \), where it is outperformed slightly by 1_B. We did not test the index-less approach 5_A.

We have further evaluated the three top-performing programs on our second evaluation environment with a different number of threads. In Figure 23, the results are shown. For all programs a higher number of threads reduces the runtime. The results show a similar behavior as when joining READS: it seems that performing a join with a small \( k \) usually is better with a small number of threads, while for larger values of \( k \) it makes indeed sense to use parallelism.

6. POST-COMPETITION ANALYSIS

The main results of our competition are shown in Figure 11. For each task and dataset we list the techniques used by the three top performing teams. The partitioning and pruning techniques of Team 1 show the best performance for three out of four problems. Only for searching our READS dataset, the acyclic word graph of Team 4 slightly outperforms Team 1's techniques.

In the following we discuss our results and relate them to existing work not covered by the competition.

6.1 Additional algorithms

We compare the results of the competition to existing tools for approximate string search. We only take into account non-dominated methods from Figure 1, for which no participant of our competition had a direct contribution. The only such non-dominated method is SSI [9]. In addition, we test two other methods: Flamingo [3], which is often used as baseline for evaluation, and Pearl [23], a prefix tree index. The results are shown in Figure 12, together with the comparison of the best three ranked programs from our competition.
Unfortunately, Flamingo has only implemented approximate search, no approximate join. We run Flamingo with the standard configuration (filters as set by the Getting-Started-example) and different length of q-grams. Index and search times are considerably longer than many of the competitors in our competition. However, note that Flamingo makes only use of one thread and the memory footprint seems to be very small. Possibly, performance of Flamingo can be further improved by additional filters. We have tested SSI only on the READS datasets. For each CITIES dataset, SSI stopped with a insufficient memory exception. This might be a bug affecting the handling of large alphabets.

The best programs from our competition outperform these tools by a factor of 1000 and more for READS-MEDIUM and a factor of 50 and more for CITIES-HUGE. In addition, we have evaluated Pearl for joining GEONAMES datasets: Even for GEONAMES-MEDIUM and \( k = 4 \), Pearl needs more than 1 hour to compute the self-join, while \( 1_A \) needs less than 2 minutes. Given the existing evaluation results from Figure 1, for each non-dominated method either one of its authors has contributed to our competition (Pass-Join, Trie-Join, PPJoin, NGPP) or the method (SSI) was shown to be way less scalable than our best programs. Therefore, we believe that our analysis represents the state-of-the-art in string similarity search and join.

### 6.2 Memory usage

We show the peak main memory usage with respect to READS-Huge in Figure 24. Programs 5_B, 5_C, and 5_D only use around 13.6 GB of main memory, followed by 3_A with 15.6 GB. The maximum amount of main memory is used by 9_A with 40.6 GB. The average main memory is 24.2 GB, which means that all the programs make use of roughly half of the main memory available. In Figure 25, the peak main memory usage for the dataset CITIES-HUGE is shown. Most of the programs show modest memory usage; the average is only 6 GB. The most main memory is used by Program 6_A: 24.7 GB, followed by 4_A and 4_B with 12-13 GB.
gram 9A only uses 0.6 GB of main memory. Thus, most of the main memory is left unused. We conjecture that it might be possible to further improve query answering times for some techniques by pre-computation of more sophisticated index structures.

### 6.3 CPU utilization

In Figure 13, the number of active threads is shown over time when searching CITIES-LARGE. The graphs of 1A, 4B, 5C, and 5D are not shown since they are very similar to 1B, 4A, 5B, and 5B, respectively. Most of the programs start preprocessing with one thread and then increase the number of threads. Program 3A is the only program which does not follow this pattern. Load scheduling of programs 1B and 4A can possibly be improved, since these programs do not make constant use of the full number of available cores. Program 4A has a long single-thread preprocessing phase; queries are answered using 16 threads.

In Figure 14, the number of active threads is shown over time when joining READS-MEDIUM with $k = 4$. The graphs of 1A, 4B, 5C, and 5D are not shown since they are very similar to 1B, 4A, 5B, and 5B, respectively. The overall join time for 1B is only few seconds, so the graph is not as stable as the other ones. For Program 4A and 5B the preprocessing phase can be clearly identified (with only one thread). Program 4A makes use of 16 threads instead of only 8. Program 9A uses 8 threads for most of the time. (only the first few seconds are run with only one thread).

### 6.4 Redundancy

The official rules allowed to serialize the same answer several times: sometimes the same result is found by different components of a search algorithm independently. In Figure 26, we analyze the redundancy in the results. The programs of Team 4 and Team 5 report answers several times (in average 4-6 times). All other programs report each answer only once (baseline 100 percent).

### 7. CONCLUSION

We believe that our evaluation gives a fairly representative picture of the state-of-the-art in string similarity search and join for different data set sizes, different alphabet sizes, and different error thresholds. Based on our datasets and competing programs, we conclude that an error rate of 20-25% pushes today's techniques to the limit. For instance, self-joining a set of 15,000,000 sequence reads of length 100 with an edit distance threshold $k = 16$ takes almost one day even for the best participant. However, the final result has more than 50,000,000 entries, which makes the usefulness of such queries in real applications questionable.

Our experiments showed that many participants used less main memory than available. The effect is conspicuous for our CITIES dataset: more than half of the competitors used less than 10 percent of the main memory. An interesting lead for future research are indexing strategies that make full use of existing main memory. Even for smaller datasets, query answering times might be further reduced by more precomputation at indexing time.

Although we have ranked programs based on search time, we have also measured indexing time separately. We found that indexing times vary a lot between implementations; in addition many programs use only one thread for indexing. Another point that could be improved as revealed by our analysis is to improve thread utilization, especially for current hardware with their quickly increasing number of cores.

It is interesting to note that the three top performing teams use different techniques. Combining these techniques, e.g. the bit-parallel LCS computation from Team 3 with the pruning techniques of Team 1, could probably reduce search and join times beyond the state-of-the-art.

### 8. ACKNOWLEDGMENTS

We thank Nikolaus Augsten for his insightful comments on a draft version of this paper. In addition, we thank Thomas Stolte for providing us Figure 1.

### 9. REFERENCES


Figure 15: Batch effect for READS-MEDIUM: Time per query for a different number of total queries (1-200,000 queries) [time in milliseconds].

Figure 16: Search times for READS-MEDIUM and different values of $k$ [time in seconds].

Figure 17: Search times for CITIES-MEDIUM [time in seconds].

Figure 18: Join times for READS and $k = 0$ [time in seconds].

Figure 19: Join times for CITIES and $k=4$ [time in seconds].

Figure 20: Join times for READS and $k = 16$ [time in seconds].

Figure 21: Join times for READS-MEDIUM on System 2 [time in milliseconds].

Figure 22: Join times for CITIES and $k=0$ [time in seconds].
Threads Progr. k=0 k=1 k=2 k=3 k=4

<table>
<thead>
<tr>
<th></th>
<th>1_A</th>
<th>0.06</th>
<th>0.30</th>
<th>0.53</th>
<th>1.83</th>
<th>8.12</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1_A</td>
<td>0.08</td>
<td>0.27</td>
<td>0.38</td>
<td>0.94</td>
<td>3.14</td>
</tr>
<tr>
<td>24</td>
<td>1_A</td>
<td>0.11</td>
<td>0.31</td>
<td>0.39</td>
<td>0.85</td>
<td>2.42</td>
</tr>
<tr>
<td>80</td>
<td>1_A</td>
<td>0.10</td>
<td>0.17</td>
<td>0.07</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td></td>
<td>1_A</td>
<td>10.40</td>
<td>10.35</td>
<td>11.15</td>
<td>17.06</td>
<td>46.00</td>
</tr>
<tr>
<td></td>
<td>5_B</td>
<td>5.76</td>
<td>4.07</td>
<td>65.28</td>
<td>760.71</td>
<td>2,353.97</td>
</tr>
<tr>
<td></td>
<td>1_A</td>
<td>0.08</td>
<td>0.27</td>
<td>0.38</td>
<td>0.94</td>
<td>3.14</td>
</tr>
<tr>
<td></td>
<td>4_A</td>
<td>10.40</td>
<td>10.35</td>
<td>11.15</td>
<td>17.06</td>
<td>46.00</td>
</tr>
<tr>
<td></td>
<td>5_B</td>
<td>5.76</td>
<td>4.07</td>
<td>65.28</td>
<td>760.71</td>
<td>2,353.97</td>
</tr>
<tr>
<td></td>
<td>1_A</td>
<td>0.11</td>
<td>0.31</td>
<td>0.39</td>
<td>0.85</td>
<td>2.42</td>
</tr>
<tr>
<td></td>
<td>4_A</td>
<td>10.40</td>
<td>10.35</td>
<td>11.15</td>
<td>17.06</td>
<td>46.00</td>
</tr>
<tr>
<td></td>
<td>5_B</td>
<td>5.76</td>
<td>4.07</td>
<td>65.28</td>
<td>760.71</td>
<td>2,353.97</td>
</tr>
<tr>
<td></td>
<td>1_A</td>
<td>0.11</td>
<td>0.31</td>
<td>0.39</td>
<td>0.85</td>
<td>2.42</td>
</tr>
<tr>
<td></td>
<td>4_A</td>
<td>10.40</td>
<td>10.35</td>
<td>11.15</td>
<td>17.06</td>
<td>46.00</td>
</tr>
<tr>
<td></td>
<td>5_B</td>
<td>5.76</td>
<td>4.07</td>
<td>65.28</td>
<td>760.71</td>
<td>2,353.97</td>
</tr>
</tbody>
</table>

Figure 23: Join times for CITIES-MEDIUM on System 2 [time in seconds].

Memory (GB)

<table>
<thead>
<tr>
<th></th>
<th>1_A</th>
<th>1_B</th>
<th>3_A</th>
<th>4_A</th>
<th>5_B</th>
<th>5_C</th>
<th>5_D</th>
<th>7_A</th>
<th>9_A</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>1_A</td>
<td>24</td>
<td>1_B</td>
<td>24</td>
<td>3_A</td>
<td>3_A</td>
<td>4_A</td>
<td>4_A</td>
<td>5_B</td>
</tr>
<tr>
<td>80</td>
<td>1_A</td>
<td>80</td>
<td>1_B</td>
<td>80</td>
<td>3_A</td>
<td>3_A</td>
<td>4_A</td>
<td>4_A</td>
<td>5_B</td>
</tr>
</tbody>
</table>

Figure 24: Peak main memory usage for READS-HUGE [memory in GB].

Memory (GB)

<table>
<thead>
<tr>
<th></th>
<th>1_A</th>
<th>1_B</th>
<th>3_A</th>
<th>4_A</th>
<th>5_B</th>
<th>5_C</th>
<th>5_D</th>
<th>6_A</th>
<th>8_A</th>
<th>9_A</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>1_A</td>
<td>24</td>
<td>1_B</td>
<td>24</td>
<td>3_A</td>
<td>3_A</td>
<td>4_A</td>
<td>4_A</td>
<td>5_B</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>1_A</td>
<td>80</td>
<td>1_B</td>
<td>80</td>
<td>3_A</td>
<td>3_A</td>
<td>4_A</td>
<td>4_A</td>
<td>5_B</td>
<td></td>
</tr>
</tbody>
</table>

Figure 25: Peak main memory usage for CITIES-HUGE [memory in GB].

Figure 26: Result redundancy: Searching READS-MEDIUM with k=4 for different number of queries (1-200,000) [redundancy in percent; 100% stands for no redundant results; 200% means that in average each result is reported twice].