Efficient Bounded Jaro-Winkler Similarity Based Search

Jan Martin Keil[1]

Abstract: The Jaro-Winkler similarity is a widely used measure for the similarity of strings. We propose an efficient algorithm for the bounded search of similar strings in a large set of strings. We compared our approach to the naive approach and the approach by Dreßler et al. Our results prove a significant improvement of the efficiency in computation of the bounded Jaro-Winkler similarity for querying of similar strings.

Keywords: Jaro-Winkler Similarity; Similarity Search; String Similarity

1 Introduction

The Jaro-Winkler similarity is a widely used measure for the similarity of strings. It was developed for the detection of duplicated persons in a dataset based on their name [Wi90]. Compared to other measures it provides both good results and fast computation [CRF03]. Nevertheless, the sequential calculation of the Jaro-Winkler similarity for the search of similar strings in large sets of strings is still a time-consuming task. Therefore, an optimized algorithm is needed for time-sensitive use cases like real time duplicate detection during data input, real time identification of named entities in input text (named entity linking), or real time fuzzy search.

We propose an optimized algorithm for the search of similar strings in a large set of strings. This work is structured as follows: In Sect. 2, we explain the Jaro-Winkler similarity and give an overview of related work, followed by the description of our approach in Sect. 3. In Sect. 4, we present the empirical evaluation of our approach. Finally, we conclude our work in Sect. 5.

2 Related Work

In this section we introduce the Jaro-Winkler similarity as well as other work on the efficient computation of this similarity measure.

[1] Heinz Nixdorf Chair for Distributed Information Systems, Institute for Computer Science, Friedrich Schiller University Jena, Germany, jan-martin.keil@uni-jena.de, https://orcid.org/0000-0002-7733-0193

doi:10.18420/btw2019-13

2.1 Jaro-Winkler Similarity

The Jaro-Winkler similarity, often wrongly called *Jaro-Winkler distance*, is a similarity measure for two strings proposed in [Wi90]. It is based on the Jaro similarity

$$\text{Jaro}(s_1, s_2) = \begin{cases} \frac{1}{3} \cdot \left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m} \right) & : m > 0 \\ 0 & : \text{otherwise} \end{cases} \tag{1}$$

where $|s_1|$, $|s_2|$ are the lengths of both strings, m is the number of matching characters, and t is the number of transpositions. Matching characters are common characters in both strings with a maximum distance of $w = \frac{\max(|s_1|, |s_2|)}{2} - 1$ [DN17; Wi90]. Some sources give an alternative definition of $w = \frac{\min(|s_1|, |s_2|)}{2}$ [CRF03]. A character matches at most one character in the other string, selected by picking the first candidate during a nested iteration over both strings. The number of transpositions t is half the number of not equal positions in the concatenated strings of all matching characters in order of original occurrence [CRF03; Wi90]. Contrary to frequent assumptions, t is not equal to the number of permutations that is required to align the order of the matching characters. For example, $t(abc, bca) = 1.5$, even though two permutations are required to align the matching characters abc and bca.

The Jaro-Winkler similarity adds a boost for equal prefixes to high Jaro similarity values:

$$\text{JaroWinkler}(s_1, s_2) = \begin{cases} \text{Jaro}(s_1, s_2) + l \cdot p \cdot (1 - \text{Jaro}(s_1, s_2)) & : \text{Jaro}(s_1, s_2) \geq b_t \\ \text{Jaro}(s_1, s_2) & : \text{otherwise} \end{cases} \tag{2}$$

l is the length of the common prefix of both strings up to a maximum l_{bound}, b_t is the boost threshold, p is the prefix scale, and $l_{bound} \cdot p \leq 1$ must holds true. Implementations typically use the values $l_{bound} = 4$, $b_t = 0.7$, and $p = 0.1$ as in the original implementation[2].

2.2 Efficient Jaro-Winkler Similarity Computation

Dreßler et al. proposed an optimized algorithm to reduce the computation effort for pairwise similarities of strings from two large sets of strings given a similarity threshold [DN17]. They reduced the number of Jaro-Winkler similarity computations by applying filters to the pairs of strings. The first filter determines an upper bound of the Jaro similarity based on the lengths of both strings. The second filter determines an upper bound of the Jaro similarity based on the maximum number of matching characters using the character histograms of both strings. This approach provides an enormous performance improvement compared to a naive implementation for the matching of two large sets of strings. Therefore, it is useful for the matching of datasets or the detection of duplicates. However, if the task is to process user input or queries, i.e. compare one or a few strings with a large set of strings,

[2] https://web.archive.org/web/19990822155334/http://www.census.gov:80/geo/msb/stand/strcmp.c

a naive implementation outperforms this approach due to its overhead. Thus, this approach is not appropriate for the processing of user input or queries.

A recently published approach by Wang et al. addresses this issue [WQW17]. They developed an index for the Jaro-Winkler similarity search that contains special string signatures. Based on a lower bound of the number of matching characters and these signatures, they select candidate strings that might be similar to a query string. Further, the order of the signatures in the index allows to abort the scan of the remaining index at a certain point.

3 Approach

We propose another Jaro-Winkler similarity algorithm, which reduces the computational effort for the search of strings (*terms*) that are similar to a single string s_1 (*query*) in a large set of strings S_2 (*terminology*) given a similarity threshold θ. The terminology will be stored in a customized *PATRICIA tree* [Mo68] that additionally stores at each node the string lengths of all subjacent leaf nodes. This enables to skip irrelevant terms by skipping whole branches of the tree. Compared to a *trie* it avoids node chains without junctions at the bottom. This reduces the number of similarity computations. The maximum distance between matching characters (w) and thereby the number of matching characters (m) depends on the lengths of both strings. Therefore, the tree will be traversed once for each length of terms in the terminology, as in List. 1. Nodes without subjacent leaf nodes with this length will be ignored. Many traversals will stop at the root node, due to of the low maximum Jaro-Winkler similarity of strings with a notable difference of lengths.

During the traversal at each node the maximum Jaro-Winkler similarity will be computed based on the query string s_1 and the known prefix s_2^* of the term strings, as in List. 2. If the maximum Jaro-Winkler similarity is less then the threshold θ, the traversal of the current branch will be skipped, as in line 11 of List. 2. This requires a function for the maximum Jaro-Winkler similarity of a string s_1 and all strings S_2^* with prefix s_2^* and length $|s_2|$. JaroWinkler, Jaro, l, m, and t depend on s_1 and s_2. Given $\max_{s_2 \in S_2^*}(\text{Jaro}) \geq b_t$, $0 \leq l \cdot p \leq 1$, and $0 \leq \text{Jaro} \leq 1$, then

$$\max_{s_2 \in S_2^*}(\text{JaroWinkler}) = \max_{s_2 \in S_2^*}(\text{Jaro} + l \cdot p \cdot (1 - \text{Jaro})) \tag{3}$$

$$= \max_{s_2 \in S_2^*}(1 - 1 + \text{Jaro} + l \cdot p - l \cdot p \cdot \text{Jaro}) \tag{4}$$

$$= \max_{s_2 \in S_2^*}(1 - (1 - \text{Jaro}) \cdot (1 - l \cdot p)) \tag{5}$$

$$= 1 - (1 - \max_{s_2 \in S_2^*}(\text{Jaro})) \cdot (1 - \max_{s_2 \in S_2^*}(l) \cdot p) \tag{6}$$

Therefore, the maximum Jaro-Winkler similarity is

$$\max_{s_2 \in S_2^*}(\text{JaroWinkler}) = \begin{cases} 1 - \left(1 - \max\limits_{s_2 \in S_2^*}(\text{Jaro})\right) \cdot \left(1 - \max\limits_{s_2 \in S_2^*}(l) \cdot p\right) & : \max\limits_{s_2 \in S_2^*}(\text{Jaro}) \geq b_t \\ \max_{s_2 \in S_2^*}(\text{Jaro}) & : \text{otherwise} \end{cases} \tag{7}$$

The function for the maximum Jaro similarity is

$$\max_{s_2 \in S_2^*}(\text{Jaro}) = \begin{cases} \frac{1}{3} \cdot \left(\frac{\max\limits_{s_2 \in S_2^*}(m)}{|s_1|} + \frac{\max\limits_{s_2 \in S_2^*}(m)}{|s_2|} + 1 - \frac{\min\limits_{s_2 \in S_2^*}(t)}{\max\limits_{s_2 \in S_2^*}(m)} \right) & : \max\limits_{s_2 \in S_2^*}(m) > 0 \\ 0 & : \text{otherwise} \end{cases} \tag{8}$$

List. 1: Search Initialization

```
1  FUNCTION search(s1,S2,θ)
2   treeRoot := tree(S2)
3   result := ∅
4   FOREACH lengths(treeRoot) AS |s2|
5    w := max(|s1|,|s2|)/2 − 1
6    s := newState()
7    s.minM := 0
8    s.minT := 0
9    s.maxL := min(4, |s1|, |s2|)
10   s.saveCommonChars1 := 0
11   s.assigned1 := boolean[|s1|]
12   s.assigned2 := boolean[|s2|]
13   s.commonChars2 := ""
14   traverse(treeRoot,s1,|s2|,s2*,θ,w,s,result)
15  RETURN result
```

List. 2: Tree Traverse

```
1  FUNCTION traverse(node,s1,|s2|,s2*,θ,w,s,&result)
2   FOREACH characters of node AS c
3    append(s2*,c)
4    updateMaxL(s1,s2*,w,c,s)
5    updateMinM(s1,s2*,w,c,s)
6    updateMinT(s1,s2*,w,s)
7   updateMaxM(s1,s2*,w,s)
8   IF |s2| = |s2*|
9    finaliseMinT(s1,s2*,s)
10  maxJWS := using Eq. (7)
        with |s1|,|s2|,s.maxL,s.maxM,s.minT
11  IF maxJWS ≥ θ
12   IF |s2| = |s2*|
13    add(result,<s2*,maxJWS>)
14   ELSE
15    FOR child ∈ children(node)
16     IF |s2| ∈ lengths(child)
17      sc := deepCopy(s)
18      traverse(child,s1,|s2|,s2*,θ,w,sc,result)
```

At each traversed node $\max_{s_2 \in S_2^*}(l)$, $\max_{s_2 \in S_2^*}(m)$, and $\min_{s_2 \in S_2^*}(t)$ will be computed. The effort can be reduced by reusing results from the parent node and updating them according to the new s_2 characters. For each new character it will be checked, if $\max_{s_2 \in S_2^*}(l)$ needs to be reduced, as in List. 3. To compute $\max_{s_2 \in S_2^*}(m)$, for each new character $\min_{s_2 \in S_2^*}(m)$ needs to be updated, as in List. 6: Each new s_2 character will be compared to the not matched s_1 characters in range. The first matching s_1 character according to reading order will be selected and $\min_{s_2 \in S_2^*}(m)$ will be updated. $\max_{s_2 \in S_2^*}(m)$ will then be computed once per node by adding the number of possible further matches to $\min_{s_2 \in S_2^*}(m)$, as in List. 7. $\min_{s_2 \in S_2^*}(t)$ can only be computed for s_1 characters that are already outside of the range of new s_2 characters, as new matching characters of s_1 might be located before earlier ones. Therefore, for each new s_2 character $\min_{s_2 \in S_2^*}(t)$ can be updated regarding s_1 characters at a new save position, as in List. 4. When s_2 is complete, t can be updated regarding the remaining characters of s_1, as in List. 5. This algorithm overestimates $\max_{s_2 \in S_2^*}(\text{Jaro})$ if t and m can not achieve their extreme values at the same time. For example, for $s_1 = \texttt{abcd}$ and

$s_2 = \mathsf{b}\text{␣␣␣}$ the values are $\max_{s_2 \in S_2^*}(m) = 4$ and $\min_{s_2 \in S_2^*}(t) = 0$, but $\max_{s_2 \in S_2^*, t=0}(m) = 3$ and $\min_{s_2 \in S_2^*, m=4}(t) = 1$.

List. 3: Computation of $\max_{s_2 \in S_2^*}(l)$

```
1  FUNCTION updateMaxL(s1,s2*,w,c,s)
2   IF |s2*| ≤ l_bound AND s1[|s2*| − 1] ≠ c
3     s.maxL := |s2*| − 1
```

List. 4: Computation of $\min_{s_2 \in S_2^*}(t)$

```
1  FUNCTION updateMinT(s1,s2*,w,s)
2   i := |s2*| − w − 1
3   IF 0 ≤ i AND i < |s1|
4    IF s.assigned1[i]
5     IF s1[i] ≠ s.commonChars2[s.saveCommonChars1]
6       s.minT := s.minT + 0.5
7     s.saveCommonChars1 := s.saveCommonChars1 + 1
```

List. 5: Final computation of t

```
1  FUNCTION finaliseMinT(s1,s2*,w,s)
2   FOR i:= max(0, |s2*|−w|) TO min(|s1|, |s2|+w)−1
3    IF s.assigned1[i]
4     IF s1[i] ≠ s.commonChars2[s.saveCommonChars1]
5       s.minT :+= s.minT + 0.5
6     s.saveCommonChars1 := s.saveCommonChars1 + 1
```

List. 6: Computation of $\min_{s_2 \in S_2^*}(m)$

```
1  FUNCTION updateMinM(s1,s2*,w,c,s)
2   FOR i := max(0, |s2*| − w − 1) TO
          min(|s1|, |s2*| + w) − 1
3    IF not(s.assigned1[i]) AND s1[i] = c
4     s.assigned1[i] = true
5     s.assigned2[|s2*| − 1] = true
6     append(s.commonChars2,c)
7     s.minM := s.minM + 1
8     BREAK
```

List. 7: Computation of $\max_{s_2 \in S_2^*}(m)$

```
1  FUNCTION updateMaxM(s1,s2*,w,s)
2   assignable1 := 0
3   FOR i:= max(0, |s2*| − w − 1) TO
          min(|s1|, |s2| + w) − 1
4     IF not(s.assigned1[i])
5       assignable1 := assignable1 + 1
6   assignable2 := |s2| − |s2*|
7   s.maxM : = s.minM + min(assignable1,
          assignable2)
```

The underlying strategy of our approach as well as the approach by Wang et al. is the early termination of the similarity computation for not similar strings. However, our approach successively approximates the similarity and extensively reuses earlier results. Conversely, the approach by Wang et al. once filters the strings by a rough upper bound of the similarity before computing the exact similarity. Both approaches skip computations for strings based on the intermediate results for other strings.

4 Evaluation

For the evaluation we used a Java implementation of each algorithm. *Our implementation* is publicly available[3] and was used in version 0.1. For the *algorithm by Dresler et al.* we used their implementation[4]. To avoid a bias, we added a few modifications[5], including the removal of a parallel execution management overhead during serial execution, and correction of bugs that skip parts of the result. While some of the changes decrease the runtime of the implementation, others increased them. However, to the best of our knowledge, the

[3] https://mvnrepository.com/artifact/de.uni_jena.cs.fusion/similarity.jarowinkler/0.1.0

[4] https://github.com/kvndrsslr/SemanticWeb-QuickJaroWinkler

[5] https://github.com/fusion-jena/QuickJaroWinkler

modifications did not add any unnecessary increase of the runtime. For the *naive algorithm* we used the Jaro-Winkler similarity implementation provided in the Apache Commons Text library[6] version 1.4. To correct the computation results we added a few modifications[7], which became part of the 1.5 release of the library. A comparison to the *approach by Wang et al.* was not possible. The approach is not described in sufficient detail in the paper to reimplement it and the implementation is not publicly available. We are in contact with the authors, though, and aim to compare the approaches in the future.

All implementations require a preparation of the terminology, like building up the PATRICIA tree, but of different extent. We distinguished between the preparation and the actual similarity computation. To evaluate our approach we tested the following hypothesis:

Hypothesis 1 *Using our algorithm will improve the efficiency of the bounded Jaro-Winkler similarity computation between few queries and prepared large sets of terms, compared to the algorithm by Dresler et al. and the naive algorithm.*

4.1 Methods

We used the Java benchmark harness OpenJDK JMH[8] to execute performance measurements of the three implementations. A collection of 1.429.572 names from the dataset "Person data" in the DBpedia dump 2016-10[9] was used as test data. We used the following measurement parameters, which cover the intended use cases: (a) The *number of queries* (10^0 to 10^5; 10^6 was skipped du to long duration), (b) the *number of terms* (10^0 to 10^6), (c) the *threshold* of the Jaro-Winkler similarity(0.91, 0.95, 0.99). (d) the *overlap* of the set of query string and term strings (*full* means that all terms are contained in the queries, if possible for the given number of queries; *half* means that half of the terms are contained in the queries, if possible; *none* means that none of the terms is contained in the queries), and (e) the *preparation* of the terminology, specifying whether the time for preparation will be contained in the measurement (*unprepared*) or not (*prepared*).

Each configuration and implementation was executed on three machines with 20 different pseudo random subsets of names for the terms and the queries, resulting in 60 executions per configuration and implementation. We measured the throughput, which is the number of executions of all queries (= one operation) per second. The usage of the throughput results in a high precision of the measurement for short running computations, but decreasing precision for longer running computations. This fits to the intended use cases. The measurements were executed on 18 machines each equipped with two Intel Xeon Scalable 6140 18 Core 2,3 Ghz processors and 192 GB memory. Parallel computation was not used to avoid measurement

[6] https://commons.apache.org/proper/commons-text/

[7] https://github.com/apache/commons-text/pull/87

[8] http://openjdk.java.net/projects/code-tools/jmh/

[9] https://wiki.dbpedia.org/downloads-2016-10

errors. The measurement code, execution scripts, analysis scripts and result files are publicly available[10].

4.2 Results

We used the Welch's t-test (unequal variances t-test) to compare the measurements, as we can not assume equal variance. First, we compared the three executions with equal configuration and of the same implementation. In 309 of 6570 cases (naive 6, Dresler 246, our 57) we found significant differences. Therefore, we hereinafter use the median values of the three corresponding execution. Then, we compared the corresponding measurements of different implementations. The overlap parameter caused only slightly differences between the comparison results. Therefore, we omit separate results. Tab. 1 shows the comparison results of the measurements of different implementations with equal configuration except the overlap parameter. Every triangle in the table represents the comparison results of 60 measurements for the column implementation and 60 measurements for the row implementation. The triangles point at the implementation with higher mean throughput. Comparisons above the diagonal involved unprepared measurements, comparisons under the diagonal involved prepared measurements. Bracketed comparisons were not significant. For example, for the naive approach and our approach and the parameters 1 term, 10 queries, threshold 0.91, and with preparation the mean throughput of the naive approach was insignificantly higher represented by ($\triangle$). Fig. 1 shows the mean measurement for the implementations with 10^6 terms, depending on the number of queries, the threshold, and the preparation. All axes are log scaled.

The results presented in Tab. 1 prove that our approach significantly improves the computation efficiency of the bounded Jaro-Winkler similarity with 100 to 10^6 prepared terms, threshold ≥ 0.91, and up to 10^3 queries, compared to the approach by Dresler et al. and the naive approach. Therefore, we accept the hypothesis. Our measurements were limited to 10^6 terms and 10^5 queries. This limitation fits to the addressed use case. The test dataset size and the time consumed by the measurement are further limiting factors. Due to the limitation, we can not provide valid results on the comparison of the approaches for larger string sets. However, the results presented in Tab. 1 indicate that the usage of our approach will improve computation efficiency for 100 or more terms in case of small query sets. Moreover, they indicate that the usage of our approach will improve the efficiency of computations with 10 up to 10^3 queries even if the terminology is unprepared. These limits will become worse by reduction of the thresholds. Due to the measurements visualized in Fig. 1d and Fig. 1f we expect that the approach by Dresler et al. will outperform our approach for larger query sets. However, this is not the use case our approach was developed for. We aim to support the search for similar strings of one or a few strings in a large set of string.

[10] https://github.com/fusion-jena/JaroWinklerSimilarityEvaluation or DOI: 10.5281/zenodo.2269909

5 Conclusions

We presented a new approach for the efficient computation of the bounded Jaro-Winkler similarity. This approach has been evaluated by comparing it with the naive approach and the approach by Dreßler et al. [DN17]. Our results prove a significant improvement of the efficiency in computation of the bounded Jaro-Winkler similarity for querying of similar strings compared to these earlier approaches. In future work, we aim to also compare our approach with the approach by Wang et al. [WQW17], depending on the availability of the implementation or a comprehensive description of the approach. Further, we provide a ready to use Java implementation of our approach for easy application and adaptation into other languages. We are convinced, that this work opens up new application fields of the Jaro-Winkler similarity.

Acknowledgments. Part of this work was funded by DFG in the scope of the LakeBase project within the Scientific Library Services and Information Systems (LIS) program. The computational experiments were performed on resources of Friedrich Schiller University Jena supported in part by DFG grants INST 275/334-1 FUGG and INST 275/363-1 FUGG. Many thanks to Frank Löffler for very helpful advice on the evaluation setup. Likewise many thanks to the three anonymous reviewers and the shepherd Ingo Schmitt for very helpful comments on earlier drafts of this manuscript.

References

[CRF03] Cohen, W. W.; Ravikumar, P.; Fienberg, S. E.: A Comparison of String Distance Metrics for Name-Matching Tasks. In (Kambhampati, S.; Knoblock, C. A., eds.): Proceedings of IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03), August 9-10, 2003, Acapulco, Mexico. Pp. 73–78, 2003.

[DN17] Dreßler, K.; Ngomo, A. N.: On the efficient execution of bounded Jaro-Winkler distances. Semantic Web 8/2, pp. 185–196, 2017, DOI: 10.3233/SW-150209.

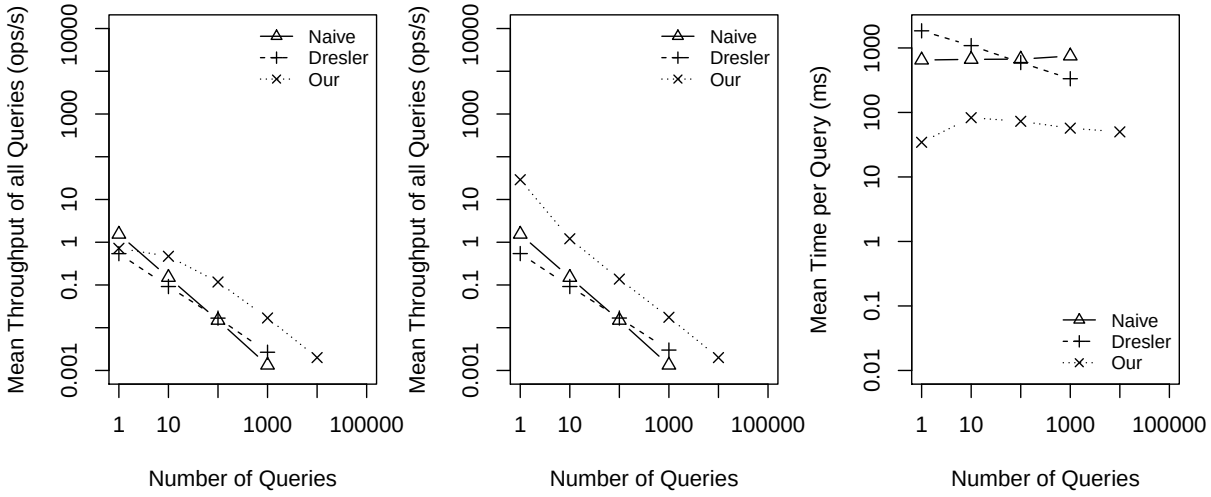
[Mo68] Morrison, D. R.: PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. J. ACM 15/4, pp. 514–534, 1968, DOI: 10.1145/321479.321481.

[Wi90] Winkler, W. E.: String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. In: Proceedings of the Section on Survey Research. American Statistical Association, pp. 354–359, 1990.

[WQW17] Wang, Y.; Qin, J.; Wang, W.: Efficient Approximate Entity Matching Using Jaro-Winkler Distance. In (Bouguettaya, A. et al., eds.): Web Information Systems Engineering - WISE 2017 - 18th International Conference, Puschino, Russia, October 7-11, 2017, Proceedings, Part I. Vol. 10569. Lecture Notes in Computer Science, Springer, pp. 231–239, 2017, DOI: 10.1007/978-3-319-68783-4_16.

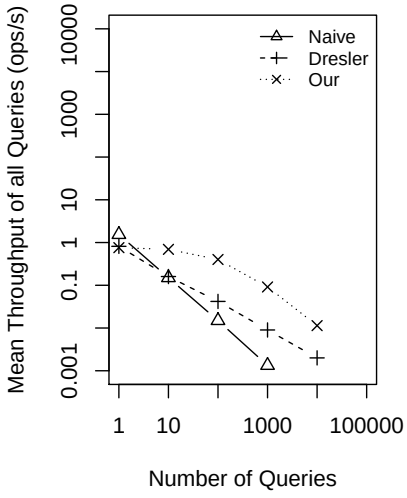
Tab. 1: Comparison results of mixed measurements with full, half, and zero overlap of terms and queries. ◀ row implementation significantly outperformed column implementation; (◀) row implementation insignificantly outperformed column implementation; △ column implementation significantly outperformed row implementation; (△) column implementation insignificantly outperformed row implementation; ? missing or too imprecise values due to long running measurements; ≈ no difference measured, due to low precision

Queries ▽		Naive										Dresler										Our									
Terms ▷		1		10		10^2		10^3		10^4		10^5		10^6		1		10		10^2		10^3		10^4		10^5		10^6	
Threshold		.91	.95	.99	.91	.95	.99	.91	.95	.99	.91	.95	.99	.91	.95	.99	.91	.95	.99	.91	.95	.99	.91	.95	.99	.91	.95	.99	.91	.95	.99
Naïve	1																														
	10																														
	10^2																														
	10^3																														
	10^4																														
	10^5																														
Dresler	1	△	△	△	△	△	(△)	△	(▼)	▼	△	(△)	▼	△	(▼)	▼															
	10	△	△	△	△	△	▼	△	▼	▼	△	△	▼	△	▼	▼															
	10^2	△	(▼)	▼	△	▼	▼	△	▼	▼	△	▼	▼	△	▼	▼															
	10^3	△	(▼)	▼	△	▼	▼	△	▼	▼	△	▼	▼	△	▼	▼															
	10^4	△	(▼)	▼	△	▼	▼	(▼)	▼	▼	△	▼	▼	?	?	?															
	10^5	△	(△)	▼	△	▼	▼	△	▼	▼	?	?	?	?	?	?															
Our	1	△	(▼)	▼	△	▼	▼	△	▼	▼	△	▼	▼	△	▼	▼	△	△	△	△	△	△	△	△	△	△	△	△	△	△	△
	10	△	(△)	▼	△	▼	▼	△	▼	▼	△	▼	▼	△	▼	▼	△	△	△	△	△	△	△	△	△	△	△	△	△	△	△
	10^2	(△)	▼	▼	△	▼	▼	△	▼	▼	△	▼	▼	△	▼	▼	△	△	△	△	△	△	△	△	△	△	△	△	△	△	△
	10^3	(△)	▼	▼	(△)	▼	▼	△	▼	▼	△	▼	▼	△	▼	▼	△	△	△	△	△	△	△	△	△	△	△	△	△	△	△
	10^4	(△)	▼	▼	(△)	▼	▼	△	▼	▼	△	▼	▼	△	▼	▼	△	△	△	△	△	△	△	△	△	△	△	△	△	△	△
	10^5	(△)	▼	▼	△	▼	▼	△	▼	▼	△	▼	▼	△	▼	▼	△	△	△	△	△	△	△	△	△	△	△	△	△	△	△
		prepared															unprepared														

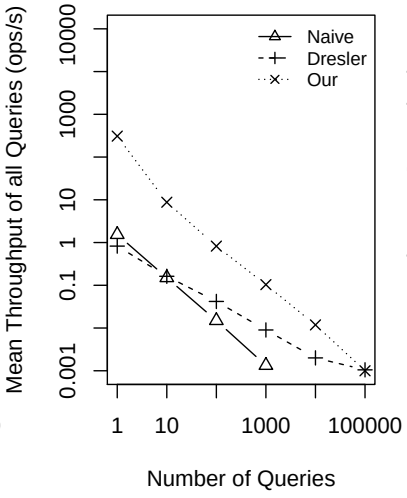


(a) Threshold 0.91, unprepared

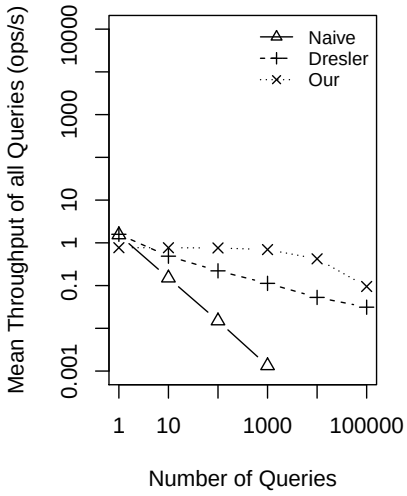
(b) Threshold 0.91, prepared



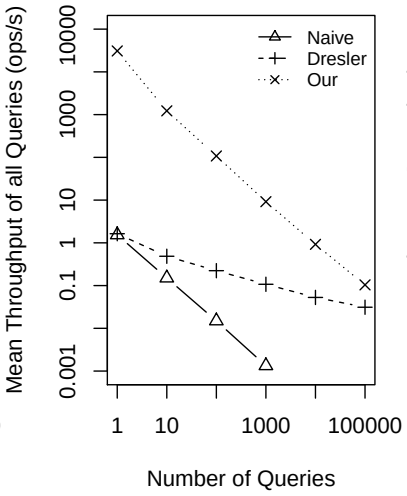
(c) Threshold 0.95, unprepared



(d) Threshold 0.95, prepared



(e) Threshold 0.99, unprepared



(f) Threshold 0.99, prepared

Fig. 1: Mean of measurements with 10^6 terms and full, half or zero coverage.