Query Processing and Optimization for a Custom Retrieval Language

Yakov Kuzin, Anna Smirnova, Evgeniy Slobodkin, and **George Chernishev**
Saint Petersburg University, Saint Petersburg, Russia
{yakov.s.kuzin, anna.en.smirnova, eugene.slobodkin, chernishev}@gmail.com

Abstract

Data annotation has been a pressing issue ever since the rise of machine learning and associated areas. It is well-known that obtaining high-quality annotated data incurs high costs, be they financial or time-related. In our previous work, we have proposed a custom, SQL-like retrieval language used to query collections of short documents, such as chat transcripts or tweets. Its main purpose is enabling a human annotator to select "situations" from such collections, i.e. subsets of documents that are related both thematically and temporally. This language, named Matcher, was prototyped in our custom annotation tool. Entering the next stage of development of the tool, we have tested the prototype implementation. Given the language's rich semantics, many possible execution options with various costs arise. We have found out we could provide tangible improvement in terms of speed and memory consumption by carefully selecting the execution strategy in each particular case. In this work, we present the improved algorithms and proposed optimization methods, as well as a benchmark suite whose results show the significance of the presented techniques. While this is an initial work and not a full-fledged optimization framework, it nevertheless yields good results, providing up to tenfold improvement.

1 Introduction

In recent years, rule-based approaches to various tasks pertaining to information extraction (IE) and natural language processing (NLP) have been "benched" by the academic community. Even ten years ago, rule-based systems were on their downfall of popularity (Chiticariu et al., 2013), and the situation does not seem to have changed now with the rise of machine learning models that are easily fine-tuned for any tasks and frameworks that provide even non-experienced users with all the necessary tools. Even the task of data annotation, which has been traditionally dealt with via manual labour, can now be simplified by annotation tools that leverage machine learning capabilities[1]. Methods such as few-shot or zero-shot learning can help deal with the problem of limited available data, and produce outstanding results in domain-independent natural language tasks.

However, rule-based approaches have held their ground in a specific area: industrial applications, especially those that require domain adaptation (Chiticariu et al., 2010b), such as biomedical information extraction (Kreimeyer et al., 2017). In settings that require high accuracy, the convenience of using an ML model can be traded off to obtain better results. However, using a *declarative* approach instead of a classical approach to IE (programs written in general-purpose programming languages intended for extraction of "hard-coded" features) adds the convenience and flexibility back. Additionally, using such approaches can help overcome issues with machine learning bias (Yapo and Weiss, 2018), unfortunate examples of which have been recorded many times.

Furthermore, the questions of performance and costs constitute a pressing issue. Using machine learning in an enterprise environment usually requires the company to both obtain expensive computational resources such as specialized GPUs and develop new ETL pipelines, given the need to protect the data that their customers provide. Furthermore, using an ML model might just not be fast or scalable enough for a business need. However, a rule-based approach to information extraction is scalable by definition, with the help of optimization and other techniques. Our project, Chat Corpora Annotator, implements a rule-based approach. It is intended for the task of data exploration and subsequent annotation. At present, its main use is exploring very long chat transcripts with extracting and annotating subsets of messages with open-domain tagsets defined by the user. A sim-

[1] A prominent example of this is prodi.gy by spaCy.

Proceedings of the 29th International Conference on Computational Linguistic, pages 61–70
October 12–17, 2022.

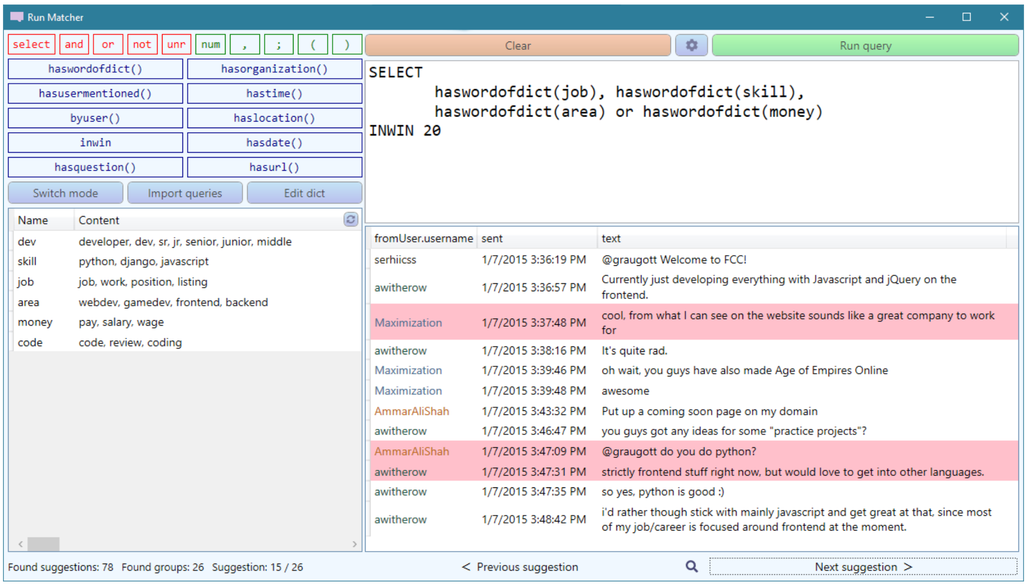

Figure 1: The Matcher interface implemented inside Chat Corpora Annotator. The left side of the window contains all of the available operators, as well as the word lists used for matching. The top frame contains a Matcher query, and the bottom frame contains its results, which can be navigated using the buttons below. The messages that match the query are highlighted in red. The query can be interpreted as extraction of mentions of job postings.

ple example of such extraction would be finding all subsets of messages in which the users of a chatroom make plans to meet in real life. Our previous paper presents the first version of the tool together with a detailed description of its intended usage (Smirnova et al., 2021). The semi-automatic rule-based extraction is implemented by our custom query language Matcher. It is a declarative, SQL-like language whose main purpose is to match over groups of messages according to the specified predicates and constraints. The supported predicates are all natural-language related, which makes the queries and their results easily interpretable. Additionally, Matcher supports complex Boolean querying and subqueries, which provide rich data exploration capabilities. Further on, we will provide the description of all supported operators and showcase several query examples on real data. Figure 1 presents the interface used to run Matcher queries.

The first version of Matcher was more of a prototype than an actual ready-to-use instrument, and, subsequently, the methods that actually retrieved and matched messages were implemented without optimization at all. Developing the second version, we ran into many performance-related issues while testing Matcher on large datasets, especially complex queries with several sub-queries, such as extremely long processing times. Therefore, we have proceeded with the decision to develop and study various query evaluation strategies for our language. Query optimization is an essential part of database query processing, so we reuse some of its core ideas in our approach. Overall, the contributions of this paper are:

1. A description of the next version of Matcher, which was extended by adding a new keyword.
2. A discussion of five query evaluation strategies which were implemented and benchmarked in the next version of CCA.
3. A benchmark suite which will provide repeatability of our experiments and may serve a basis for further studies concerning optimization of such queries.

The rest of this paper is organized as follows: Section 2 contains an overview of related work, Section 3 provides a description of our query language, Section 4 describes the algorithms that we propose for query optimization, and Section 5 and Section 6 describe the benchmark and showcase the experimental results. Finally, we give some concluding remarks in Section 7.

2 Related Work

2.1 Rule-Based Systems

Probably one of the most prominent examples, IBM's SystemT (Li et al., 2011), is an information extraction system that implements AQL — a declarative rule language that is intended for extracting structured information from unstructured documents. It is similar to SQL in syntax, and its output is presented as an SQL `view`. Below we will briefly discuss its query optimization approach. An interesting development worthy of mentioning is the rule-based NERL language (Chiticariu et al., 2010b) built on top of SystemT and intended for customized named entity recognition.

A different declarative approach is implemented in the DeepDive system (Zhang et al., 2017; Shin et al., 2015). They employ a language based on SQL and Datalog in order to facilitate the development of declarative programs for information extraction and subsequent use of this information, such as constructing knowledge bases.

Odinson (Valenzuela-Escárcega et al., 2020) is one of newest rule-based information extraction frameworks, presented in 2020. It features a system for annotating and essentially constructing a custom knowledge base, and a declarative pattern query language which is used for extracting information out of them. The language has the capabilities to run over not only tokens and token features, but graph-like annotations as well (such as syntactic dependencies). Internally, Odinson is based on a custom Lucene index, specifically implemented to index as much information about annotated documents as possible. This provides a large share of runtime optimization. Additionally, since not everything can be indexed, Odinson also contains a query compiler that optimizes the queries that involve syntax annotations, compiling them into a graph traversal pattern. The authors state that due to their optimizations, Odinson is 150,000 times faster than its predecessor Odin.

GATE (Cunningham et al., 2002) is a well-known IE system/framework which was first released in 2002. It features a possibility to construct annotators with JAPE (Java Annotation Patterns Engine), which is an imperative rule-based language adhering to the CSPL (Common Pattern Specification Language) standard. As far as we know, approaches based on CSPL cannot be optimized, as they produce a finite state transducer and have a set rule execution order.

Finally, another notable information extraction system is the UIMA Ruta framework (Klügl et al., 2016). It features an expressive matching language that allows building concise representations of matching rules. It is not declarative, similarly to GATE. However, the authors state that their language does not suffer from the drawbacks of CSPL, as it supports variable execution order, but they do not touch on optimization.

2.2 Query Optimization

Overall, query optimization is a well-developed and well-studied area. Starting out in 1979 with System R's optimizer (Selinger et al., 1979), studies of optimization allowed optimizers to become a standard and indispensable feature in all industrial DBMSes (Özsu and Valduriez, 2011). Some of the most prominent works of this area concern the Starburst and the Volcano optimizers. Overall, over the years optimization has accumulated a rich set of concepts and principles such as selectivity, interesting orders, minimization of intermediate results, data sketches (histograms) and many more.

However, query optimization for lesser-known purposes, such as query languages intended for information extraction, has not been properly explored. Further on, we will try to provide an overview of existing solutions. In their 2013 article (Chiticariu et al., 2010a), the authors of SystemT present an algebraic query optimizer for AQL. Unlike its main competitors, CSPL-based languages that use cascading grammars, AQL does not place evaluation order restrictions on its operators. This opens up the fundamental possibility of constructing an operator graph, and furthermore, many operator graphs for a single query, which in turn makes it possible to select the best one. The authors formally prove that the CSPL grammars cannot produce a finite state transducer that is faster than any algebraic graph, and compare the performance of SystemT and GATE on a rule-based Named Entity Recognition task. Their system won in both throughput and required memory.

The SQoUT project (Jain et al., 2009a) was a system that allowed its user to run structured queries over relation tuples extracted from natural language texts. The authors have implemented a full-fledged cost-based query optimizer for SQL over a database that stores such relation tuples. In their later article(Jain et al., 2009b), they consider join optimization for their system, focusing not only on performance, but on output quality, because in such a task it is critical: i.e., the relation tuples cannot be invalid. Their optimizer chooses between three join algorithms: Independent Join, Outer/Inner Join, and Zig-Zag Join. The authors provide a thorough evaluation of both performance and quality of output, and come to the conclusion that their optimizer is effective.

Finally, in their 2012 article, El-Helw et al. (El-Helw et al., 2012) introduce the concept of *extraction views*: database views whose data is obtained by running information extractors on specific document collections. The authors state that any extractor such as GATE or UIMA can be used for this purpose. They introduce a system that integrates SQL with such extractors, which can be used to build special tables out of tuples extracted from unstructured documents. Additionally, they provide a cost-based optimizer for SQL queries over such tables, which supports visualization of the used execution plan.

To the best of our knowledge, these are the only papers that concern optimizing queries for information extraction systems. However, none of them were intended for the task that we have formulated: extracting subsets of messages out of short text datasets, such as raw dumps of chats and tweets. They could not be adapted for the task either, as they are focused strictly on extracting pre-specified information, such as, for example, detecting phone numbers in a set of email documents. Whereas our approach is more oriented towards *discovery* of information that may match a specific, but rather loose pattern. This is prompted by the nature of chats and tweets, as they are often entangled and noisy, and traditional IE tools may fall short in the task we propose. At the same time, considering their optimization, such IE systems are necessary for both academic and industrial community and ensuring their performance is of priority. Therefore, reusing query optimization techniques from the database domain looks like a promising approach.

3 Matcher Query Language

Matcher query language consists of a basic set of rule-based matching operators, which can be combined into a *matcher* with Boolean operators, and which, in turn, can be combined into *matching groups* with commas. Furthermore, Matcher also contains a restriction operator `INWIN` and the `UNR` modifier, which will be explained below. The formal query syntax of Matcher is as follows:

```
query = SELECT body window
body = query_seq | restriction_seq
window = | INWIN N

query_seq =
  (query)
  | (query) ; query_seq

restriction_seq =
  restriction_seq_body
  | restriction_seq_body UNR

restriction_seq_body =
  restriction
  | restriction, restriction_seq

restriction =
  restriction AND restriction
  | restriction OR restriction
  | (restriction)
  | NOT restriction
  | condition
```

Here, bold denotes language keywords and regular denotes nonterminal symbols. Let us consider the language in a bottom-up fashion.

Formally, a *matcher* is a template that is matched against a single message in chat history. It consists of a Boolean expression which is evaluated for each message, and if it equals True, then the message is added to the output. Therefore, a *matching group* is a set of *matchers*, each of which provides a single message for the output. In the formal syntax, a *matcher* is described by the `restriction` nonterminal symbol, an individual rule-based matching operator by `condition`, and a *matching group* by the `restriction_seq`.

The currently the available set of rule-based matching operators is the following:

- `haswordofdict(dict)` matches messages that contain any word from a pre-specified named list;
- `hasdate()`, `hastime()`, `haslocation()`, `hasorganization()`, `hasurl()` match messages that contain tokens with the respective Named Entity annotations[2];
- `hasusermentioned(user)` matches messages that contain a username mention in the text field;
- `byuser(user)` matches messages that contain a specified username in the user field;
- `hasquestion()` matches messages that contain at least one question-like sentence.

[2]Our system obtains NER annotations via external integration with a running CoreNLP instance.

Matcher relies on a simple data schema consisting of three fields: a username-like field, a date-like field, and a text-like field. CCA focuses on chat datasets, and we believe that this is a minimal restriction on such data.

Next, the body nonterminal symbol specifies two admissible types of queries: *matcher*-only and subquery-only. The first one consists only of *matchers* separated by commas. The second one requires nested `SELECT`s (separated by semicolons) inside the parent `SELECT`.

Matcher-only queries are intended for simple use-cases, while queries with subqueries provide flexibility by allowing to express complex patterns, constructing them in a bottom-up fashion. Such queries concatenate results of individual subqueries while checking additional restrictions such as distance between them and ordering. Matcher supports subqueries of arbitrary depth. Use-case examples are provided in Section 5.

The `window` nonterminal describes an optional `INWIN` clause. It provides a way to explicitly restrict the length of the window in which all matched messages should fit. For example, query `SELECT hasdate(), haswordofdict(meeting) INWIN 20` means that in each returned set of messages, the two messages that conform to the *matchers* must not be further away than 20 messages from each other. If not specified, this length is implicitly restricted to 50 on the language implementation level for performance reasons.

The current version of Matcher has been extended with a special `UNR` clause, which significantly improves the expressiveness of the language. It is a modifier that removes the match order constraint on *matching groups*. Without this clause, the outputs of each *matching group* are ordered according to their order in the query.

4 Query Processing

4.1 Basics

The goal of our query processor is to construct a list of answers, each answer being a list of integer message ids. Thus, all operations are performed on integer lists (groups) or lists of integer lists (group lists).

Due to the space constraints we will not present the pseudocode of algorithms, but we will sketch out the ideas behind them instead.

In general, Matcher's query evaluation consists of four phases, which correspond to functions in the source code:

1. **VisitQuery**. It is the first function that handles a submitted query. If there are subqueries in it, it obtains their results and then performs merging, calling **MergeQueries** which checks order and `INWIN` requirements. If there are no subqueries, i.e. the query contains only a *matching group*, then **VisitRestrictions** is called. After this, the output of **VisitQuery** is constructed by eliminating duplicates and sorting final message lists by their first message position. Note that **VisitQuery** is a recursive function that is run for each subquery.

2. **VisitRestrictions**. It obtains a group list in which the i-th group contains all messages that conform to the i-th *matcher* (the entire Boolean formula) and sorts it if necessary. This phase also handles the `UNR` clause by generating all possible permutations of the groups. This method calls **VisitCondition** to obtain message ids that conform to individual conditions.

3. **VisitCondition**. It queries the transcript to extract all messages that conform to a single predicate (`haswordofdict`, `byuser`, `hasdate`, `hastime`, etc).

4. **MergeRestrictions**. This function accepts a group list, where each list corresponds to an individual restriction. Items of these lists are message ids that conform to the respective Boolean formula. The output is another group list, but each group corresponds to an answer. Thus, **MergeRestrictions** constructs this list by taking ids from different input groups while checking the `INWIN` clause.

Our initial experiments demonstrated that there are two most expensive functions which needed to be optimized — **VisitCondition** and **MergeRestrictions** (see Fig. 4). The first one works by issuing calls to Lucene, which stores indexed chat transcript data. There are many possible straightforward methods, e.g. implementing reuse or setting up caching, tuning Lucene, using another storage layer and so on. In terms of DBMS query processing and optimization, this part is similar to the access method selection problem. On the other hand, optimizing **MergeRestrictions** resembles join optimization and it is trickier than **VisitCondition**.

For this paper, we have decided to develop several different strategies for optimizing this part. Let us consider in detail how a Matcher query is evaluated.

Query evaluation starts with the body of the main SELECT clause. Any SELECT can consist of either a sequence of nested subqueries or a sequence of *matchers*, i.e. a matching group.

If the body consists of n *matchers*, we find the ids of messages that satisfy the respective conditions, obtaining n groups of messages. Let us denote this stage with an asterisk (*). If the `UNR` modifier is used, all possible permutations of the obtained groups are generated, and if not, a single order (which is specified in the query) is considered. For each group order (that is, for each permutation), we merge the groups taking into account order constraints and the `INWIN` condition, i.e., we create lists of n ids, each element of which is an element of some group. We check the uniqueness of these lists, deleting those that are already in the resulting list, and add the rest to the output of the current query or subquery.

However, if the body consists of k subqueries, we process each subquery and merge the obtained lists, taking into account order constraints and the `INWIN` condition, obtaining a group list sequence. Merging produces lists that contain k groups, each of which is an element of the resulting list of the corresponding subquery. Thus, this is recursion — we process sequences of subqueries until we reach a subquery that does not have its own subqueries.

Now, for the **MergeRestrictions** phase we propose several different algorithms, namely:

1. N+NS: naive, no sort;
2. N+S: naive, with sort;
3. P+NS: position-based, no sort;
4. P+S: position-based, with sort;
5. H+S: histograms, with sort.

The first part of their name encodes method and shows whether the groups to be merged have been sorted at the (*) stage (i.e., when they were obtained).

4.2 N+NS: naive, no sort

The N+NS algorithm is the most basic version of all considered algorithms. It is a recursive algorithm which at first selects the first group and starts iterating over its values, launching itself for each individual value. This value will be the beginning of an answer, which is a list. Each launched instance has this partial answer as the first parameter and the remaining groups as the second. On subsequent recursion steps, the algorithm tries to add the next message id (taking it from the first group, out the remaining ones) to the partial answer. For this, it is necessary to check: 1) whether the id of previous message is smaller than the id of the current one, and 2) whether the `INWIN` restriction holds. If there are no suitable message ids in the considered list, then this recursion branch terminates. Thus, at each recursion step, the partial answer grows by a single id until all groups are checked.

4.3 N+S: naive, with sort

The N+S algorithm is the same, except that it leverages the fact that the contents of lists to merge are sorted. Thus, it makes possible to greatly reduce the number of recursive paths to traverse.

There is a number of differences from the N+NS algorithm. Firstly, it sorts each obtained group at the (*) stage in ascending order. Next, on the second and all subsequent recursive steps it conducts different checks to test whether the considered id can participate in the result.

More specifically, if the current message id goes beyond the `INWIN` restriction, further processing of the next elements of the current group stops. Indeed, due to the ascending order of the groups, all subsequent messages of this group definitely cannot fit into the window.

Such approach allows to early terminate the evaluation, which will have a good effect on the overall performance. However, sorting will incur additional costs, which should be taken into account and which we will experimentally evaluate.

4.4 P+NS: position-based, no sort

The next algorithm is built around the following idea: we should start merging groups from the smallest ones (in terms of their size) and finish with the largest one. Using this approach, we can reduce the number of recursive branches which do not satisfy the `INWIN` and order requirements. At the same time, the need of a novel approach to checking order requirements arises. The idea is illustrated in Figure 2. When we add an id to the resulting list we have to check that the id of previous message is smaller and that id of the next message

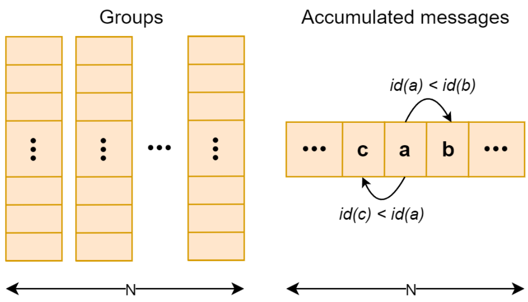

Figure 2: The P+NS algorithm

is greater (if they have been already placed). We call this class of algorithms position-based.

4.5 P+S: position-based, with sort

Similarly to P+NS, this algorithm exploits the idea of merging smallest groups first. But at the same time it relies on the sorting idea of the N+S algorithm. Thus, the P+S algorithm sorts each resulting group at the (*) stage, and at the start the algorithm is supplied with the number of messages in each group.

Then, similarly to the N+S, it is possible to prune a large number of recursive branches that do not satisfy the order and `INWIN` restrictions.

4.6 H+S: histograms with sort

The next stage in the development is the H+S algorithm, which uses equi-width histograms (Ioannidis, 2003) of message distribution. The idea is the following: unlike all previous algorithms which merged all groups at-a-time we merge groups two at-a-time.

At the same time, we try to merge groups that will result in as few intermediate results as possible first. This resembles the idea of classic optimization of a join sequence in SQL query processing, where the size of intermediates is reduced too. In order to estimate the sizes, we employ equi-width histograms, which are constructed in advance.

The algorithm itself is as follows. At first, we sort groups by their sizes. Then we iterate over triples of groups and try to assess the benefit for performing local permutations on them if their sizes are close enough. In our experiments (see Section 6) it was shown that sorting groups by their sizes (position-based approaches) already results in formidable improvement, therefore we should build our next algorithm upon this idea.

Assessing benefits of permutations is done in the following way. Suppose that we have groups A, B, C which we have to merge, and they are of similar size. We consider the following three permutations: ((AB)C), ((BC)A), and ((AC)B) which represent different evaluation orders. Each of them is assessed by "intersecting" histograms of the central part. After the intersection, we estimate the size of the intermediate result using obtained histograms. Then we select the evaluation order that corresponds to the smallest histogram.

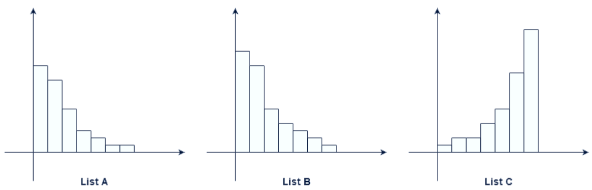

Figure 3: Distribution Example

Consider the example presented in Figure 3. Suppose that we have to merge A, B, and C lists with the message distributions as shown in the figure. It is evident that it is better to merge A and C (B and C) first than A and B. This way we will discard a lot of intermediates as soon as possible.

This approach will not provide performance improvement if there is an identical data distribution in all considered groups. But at the same time it will not incur a significant overhead except the histogram construction phase, which can be done in advance.

This is a prototype of a full-fledged optimizer intended to demonstrate its viability in our setting.

5 Benchmark

To ensure repeatability we have created a benchmark which consists of four queries and a message set. This message set contains the first 1 million messages from the freeCodeCamp Gitter Chat dataset, available on Kaggle[3]. Its Lucene index takes around 100MB of disk space.

Listing 1: Query 1 (Q1)

```
SELECT
  hasword(job), hasword(code),
  hasusermentioned(Kadams223)
UNR INWIN 40
```

The Q1 query matches a group of three messages that have words related to jobs, words related to code and a mention of a specific username. It can

[3] https://www.kaggle.com/datasets/freecodecamp/all-posts-public-main-chatroom

be interpreted as extracting a discussion that the specified user has been actively partaking in, receiving many replies. This is a simple query that illustrates the concept behind the UNR clause. The results of the *matchers* are not ordered in accordance to their order in the query, but the whole group should fit into a window of 40 messages. Note in that the following listings haswordofdict was shortened to hasword for better readability.

Listing 2: Query 2 (Q2)

```
SELECT
  hasword(job), hasword(skill),
  hasword(skill), hasword(area),
  hasword(money)
INWIN 40
```

Q2 extracts a group of five messages that discuss coding job listings with the mentions of the area of the job, required skill and the salary. This is a simple query with no subqueries, however, it has many *matchers* and a window length of 40 messages in which the extracted group should fit.

Listing 3: Query 3 (Q3)

```
SELECT
  (SELECT
    hasword(job), hasword(skill),
    hasword(code), byuser(Lumiras)
    INWIN 60
  );
  (SELECT
    byuser(Lumiras) AND hasword(issue)
  )
INWIN 200
```

The Q3 query extracts the following information: a discussion of job search and coding languages in which user Lumiras took part, and it is followed by an issue alert by the same user. This query contains two subqueries, the first of which has a medium-sized window, and the second one has a Boolean AND which specifies that the output of this *matching group*, containing a single message, must be by the specified user and contain a reference to an issue.

Listing 4: Query 4 (Q4)

```
SELECT
  (SELECT
    hasword(job), hasword(skill),
    hasword(code), byuser(Lumiras)
  );
  (SELECT
    hasword(job), hasword(skill),
    hasword(code), byuser(odrisck)
    INWIN 40
  );
  (SELECT
    hasword(job), hasword(skill),
    hasword(code), byuser(odrisck)
    INWIN 40
  )
INWIN 300
```

Q4 extracts three message groups, each of which discusses job search and coding languages. User Lumiras participates in the first group, and user odrisck takes part in the second and the third. This query contains three similar subqueries and a large window for the groups to fit in.

6 Experiments and Discussion

Experimental evaluation was conducted on a PC with the following characteristics: 8-core Intel®Core™ i7-11800H CPU @ 2.30GHz, 16 GB RAM, running Windows 10.0.19042.1645 (20H2/October2020Update).

The current version of CCA is implemented in C# (.NET 6, WPF) using the following libraries: Antlr4, Lucene.Net 4.8.0, Newtonsoft.Json, and SoftCircuits.CsvParser. To obtain accurate measurements, we have used BenchmarkDotNet v0.13.1[4]. It was run with default parameters, a confidence interval of 99.9% was calculated by the benchmark (as a different number of runs was used in each case), and we manually checked that relative error was less than 2%. Our source code is available publicly[5].

In our first experiment we have compared the four initial versions of the algorithm: N+NS, N+S, P+NS and P+S. For this, we used the first four queries of our benchmark. The overall results are presented in Table 1.

To provide better insights into the results, we have also constructed a stacked barchart presented in Figure 4. It shows four approaches each depicted by its own bar. Each bar is divided into parts that correspond to the contribution of each method.

Our experiments have clearly demonstrated that all of the proposed strategies are superior in performance to the basic N+NS. Overall, P+S is the most efficient approach, which noticeably beats N+S and N+NS, and to a lesser extent P+NS. It was possible to obtain more that 10x speedup for a subset of queries.

The next observation is that sorting of group elements is almost free in terms of time. Sorting is conducted inside the **VisitRestriction** function and

[4]https://github.com/dotnet/BenchmarkDotNet

[5]https://github.com/yakovypg/Chat-Corpora-Annotator

Query	Msgs	N+NS	Impr	N+S	Impr	P+NS	Impr	P+S	Impr
Q1	1M	2253.5	-	1232.8	1.8x	209.4	10.8x	**207.0**	10.9x
Q2	1M	4260.6	-	1954.6	2.2x	530.6	8.0x	**419.1**	10.2x
Q3	1M	4354.1	-	2248.0	1.9x	408.1	10.7x	**379.2**	11.5x
Q4	1M	11615.2	-	6193.5	1.9x	2016.7	5.8x	**1318.6**	8.8x
Total	—	22483.4	-	11628.9	1.9x	3164.8	7.1x	**2323.9**	9.7x

Table 1: Overall results

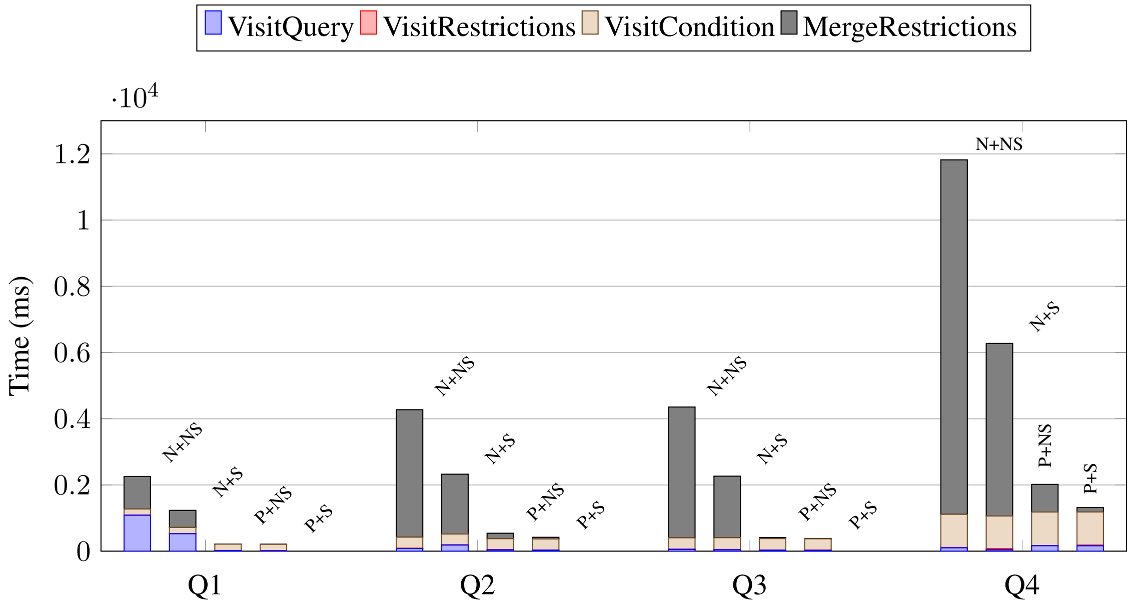

Figure 4: In-depth results

the figure shows that its presence has almost no impact on the bar height.

Listing 5: Query 5 (Q5)

```
SELECT
  byuser(sludge256),
  byuser(sludge256),
  byuser(trisell) OR byuser(seahik) OR
  byuser(odrisck) OR byuser(jsonify) OR
  byuser(cerissa) OR byuser(mykey007) OR
  byuser(AhsanBudhani) OR
  hasusermentioned(seahik)
INWIN 50
```

The second experiment concerned out last strategy, H+S. H+S does not always beat P+S, but it never loses to it, either. We have added the Q5 to our benchmark as an example on which H+S beats P+S: it takes 1479 ms compared to 1865 ms, thus providing 25% improvement.

7 Conclusion and Future Work

In this paper, we have presented and described our custom query language Matcher, intended for exploration and annotation of large natural language datasets such as chat transcripts. The main body of our work consisted in optimizing one of Matcher's execution stages that deals with merging the results of individual parts of the query, which in essence resembles join optimization. We have created five algorithms and a benchmark of five queries to test them against. We have not presented a proper optimizer, but a collection of simple techniques that nevertheless yield surprisingly good results, providing up to 10x improvement. All this warrants further investigation. Concerning our future work, the first evident direction is to implement a proper cost model, design rules for enumerating plan space, employ more sophisticated statistics and estimators of intermediate result sizes. Next, looking at graphs for the P+S algorithm, one may notice that for Q4, Lucene index access has become the most costly part and thus, it is necessary to address this. There are many approaches to optimizing Lucene index access which can be investigated. Finally, note that Matcher itself does not support variables so far, however, it is a necessary feature and it will have impact on optimization.

Acknowledgments

We would like to thank Kirill Smirnov for his comments on statistics for result processing.

References

Laura Chiticariu, Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan, Frederick Reiss, and Shivakumar Vaithyanathan. 2010a. SystemT: An Algebraic Approach to Declarative Information Extraction. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 128–137, Uppsala, Sweden. Association for Computational Linguistics.

Laura Chiticariu, Rajasekar Krishnamurthy, Yunyao Li, Frederick Reiss, and Shivakumar Vaithyanathan. 2010b. Domain Adaptation of Rule-Based Annotators for Named-Entity Recognition Tasks. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1002–1012, Cambridge, MA. Association for Computational Linguistics.

Laura Chiticariu, Yunyao Li, and Frederick R. Reiss. 2013. Rule-Based Information Extraction is Dead! Long Live Rule-Based Information Extraction Systems! In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 827–832, Seattle, Washington, USA. Association for Computational Linguistics.

Hamish Cunningham, Diana Maynard, Kalina Bontcheva, and Valentin Tablan. 2002. GATE: an Architecture for Development of Robust HLT applications. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 168–175, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Amr El-Helw, Mina H. Farid, and Ihab F. Ilyas. 2012. Just-in-time information extraction using extraction views. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 613–616, New York, NY, USA. Association for Computing Machinery.

Yannis Ioannidis. 2003. The history of histograms (abridged). In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, page 19–30. VLDB Endowment.

Alpa Jain, Panagiotis Ipeirotis, and Luis Gravano. 2009a. Building query optimizers for information extraction: the SQoUT project. *ACM SIGMOD Record*, 37(4):28–34.

Alpa Jain, Panagiotis G. Ipeirotis, AnHai Doan, and Luis Gravano. 2009b. Join Optimization of Information Extraction Output: Quality Matters! In *2009 IEEE 25th International Conference on Data Engineering*, pages 186–197. ISSN: 2375-026X.

Peter Klügl, Martin Toepfer, Philip-Daniel Beck, Georg Fette, and Frank Puppe. 2016. UIMA Ruta: Rapid development of rule-based information extraction applications. *Natural Language Engineering*, 22(1):1–40.

Kory Kreimeyer, Matthew Foster, Abhishek Pandey, Nina Arya, Gwendolyn Halford, Sandra F Jones, Richard Forshee, Mark Walderhaug, and Taxiarchis Botsis. 2017. Natural language processing systems for capturing and standardizing unstructured clinical information: A systematic review. *Journal of Biomedical Informatics*, 73:14–29.

Yunyao Li, Frederick Reiss, and Laura Chiticariu. 2011. SystemT: A Declarative Information Extraction System. In *Proceedings of the ACL-HLT 2011 System Demonstrations*, pages 109–114, Portland, Oregon. Association for Computational Linguistics.

P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, SIGMOD '79, pages 23–34, New York, NY, USA. Association for Computing Machinery.

Jaeho Shin, Sen Wu, Feiran Wang, Christopher De Sa, Ce Zhang, and Christopher Ré. 2015. Incremental Knowledge Base Construction Using DeepDive. *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, 8(11):1310–1321.

Anna Smirnova, Evgeniy Slobodkin, and George Chernishev. 2021. Situation-based multiparticipant chat summarization: a concept, an exploration-annotation tool and an example collection. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing: Student Research Workshop*, pages 127–137, Online. Association for Computational Linguistics.

Marco A. Valenzuela-Escárcega, Gus Hahn-Powell, and Dane Bell. 2020. Odinson: A fast rule-based information extraction framework. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 2183–2191, Marseille, France. European Language Resources Association.

Adrienne Yapo and Joseph W. Weiss. 2018. Ethical Implications of Bias in Machine Learning. In *HICSS*.

Ce Zhang, Christopher Ré, Michael Cafarella, Christopher De Sa, Alex Ratner, Jaeho Shin, Feiran Wang, and Sen Wu. 2017. DeepDive: declarative knowledge base construction. *Communications of the ACM*, 60(5):93–102.

M. Tamer Özsu and Patrick Valduriez. 2011. *Principles of Distributed Database Systems*, 3rd edition. Springer Publishing Company, Incorporated.