Towards an Algebraic Approach for Corpus Queries

Niklas Deworetzki[1] **and Peter Ljunglöf**[1,2] **and Nicholas Smallbone**[1,2]
[1]Department of Computer Science and Engineering,
Chalmers University of Technology and University of Gothenburg
[2]Språkbanken Text, University of Gothenburg
nikdew@chalmers.se, peter.ljunglof@gu.se, nicsma@chalmers.se

Abstract

Analysis of text corpora involves the use of specialised corpus search tools, capable of handling huge amounts of annotated text. The extent to which these tools apply optimisations to reduce query execution times is as diverse as the tools themselves. We argue that the development of a corpus algebra, similar to relational algebra in relational database systems, is a valuable foundation to improve corpus query optimisation. We demonstrate a query optimisation approach based on algebraic transformations, which vastly reduces query execution times.

1 Introduction

Text corpora and their analysis form an important cornerstone of computational linguistics and natural language processing. As annotated and searchable collections of text, spanning up to billions of words, they provide a valuable basis of data for researchers. However, the size of a corpus and the complexity of its annotations also form one of the difficulties in analysis. Corpora are analysed by searching for individual tokens, token sequences or structures matching some given criteria of interest. The amount and distribution of matches in conjunction with metadata like author and time of creation of a text can lead to valuable insights for researchers. Finding all matches, especially when criteria describe a complex relationship of different tokens and attributes, requires a considerable amount of computational resources. With response times for corpus queries in the order of minutes, creating and refining queries quickly becomes a cumbersome task.

1.1 Challenges of Query Execution

In order to highlight the challenges of executing corpus queries efficiently, we want to provide a small example. Consider the text in Figure 1, which shows a single sentence from the BNC corpus (BNC Consortium, 2007). Each token has been annotated with its part of speech and the syntactic relation it has with other words in the sentence. The following query could be used to try and find all subject or object noun phrases in a sentence:

```
[pos = DET|NUM|PRON] [pos = ADJ|NOUN]*
      [deprel = .*(SU|O)BJ]
```

It matches a token whose part of speech is *determiner* or *number* or *pronoun*, followed by a possibly empty sequence (denoted by the $*$-operator) of *adjectives* and *nouns*, followed by a token that acts as a *subject* or *object*.

While this query is not particularly complex, it is not trivial to execute it efficiently. Typical algorithms and strategies used for text search, such as suffix arrays (Manber and Myers, 1990), are difficult to apply as a corpus stores not only text, but also annotations associated with it.

Instead, specialised corpus search tools are used to access to the annotated data in a corpus. The different query languages provided by these tools are often only defined ad hoc by providing the tool's implementation as an interpreter. As a consequence, it is not clear how different features might

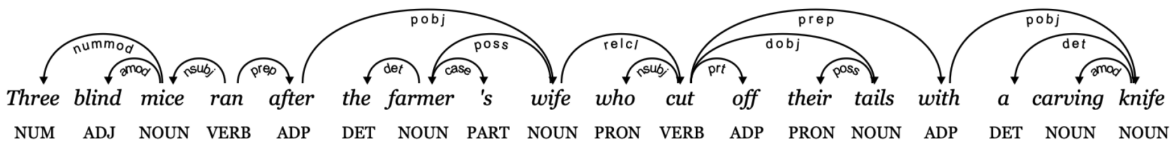

Figure 1: Text annotated with part-of-speech and syntactic relations from the BNC corpus (BNC Consortium, 2007).

interact, so it is difficult for tool authors to optimise queries correctly, and users aiming to improve the execution time of a query have to rely on manual tuning using their knowledge of a tool's inner workings. Additionally, optimisation techniques implemented by one tool can not easily be adopted by other tools.

1.2 Motivation

A look at relational database systems provides a potential approach to improve query execution times. There, query execution engines rely on solid mathematical foundations to perform automated optimisations on queries. We argue that creating an algebra for corpus queries enables similar optimisation techniques when querying corpora, which in turn speeds up execution of corpus queries and reduces the resources required by corpus tools. Our goal is to create a formalised corpus query language based on a sound *corpus algebra*. In this paper, we explain why we consider a corpus algebra a necessary foundation for corpus tools and present first results on optimised query execution using it.

2 Existing Technology

Existing query engines, languages and technologies use vastly different approaches to address the requirements of their specific domains. This section provides an overview of existing approaches for processing queries in the context of corpora, databases and text retrieval. In all cases, a query describes a result set in terms of properties a user is interested in. The means of finding that result set are not specified. This leaves room for a query engine to choose an efficient execution strategy.

2.1 Corpus Query Engines

The straightforward way of searching in a corpus is to use the query as a filter while linearly iterating through all text positions. This strategy works for smaller corpora, but for large corpora with billions of tokens, other strategies are required to answer queries without long waiting times.

Existing corpus search engines have two main approaches to making corpus search efficient: (1) they use inverted indexes, which is a technique mainly used in the field of information retrieval, or (2) they convert the corpus into a database and convert corpus queries into equivalent database queries to retrieve the results.

Using inverted indexes: Text retrieval, the branch of information retrieval that considers finding relevant parts in free-text documents, exhibits a similar goal to corpus searches. It appears therefore only natural that inverted indexes, which are used to find locations of a query string within free-text documents, are also used for corpus search by e.g., Evert and Hardie (2011), Diewald and Margaretha (2016), Meurer (2020), and Ljunglöf et al. (2024). These engines compile a complex query into one or more basic searches in statically compiled inverted indexes. Results from these searches are then combined and further refined, as available indexes might not be sufficiently specific to fully answer a query on their own. Refinement of result candidates is done by individually checking each candidate against the query's criteria. Depending on the query, this refinement requires a considerable amount of execution time.

Using database engines: Another approach is to convert the corpus, its annotation and further information into a format processable by an existing database engine. Corpus queries have to be compiled into queries for the chosen engine, which is then responsible for accessing the data in an efficient manner. Commonly advocated for this approach are relational database systems (e.g. Davies (2005), Krause and Zeldes (2016), Kleiweg and van Noord (2020) and Schaber et al. (2023)) or the *Apache Lucene* search engine[1] (e.g. Bingel and Diewald (2015), Ghodke and Bird (2012) and Luotolahti et al. (2017)). This approach can be quite efficient, if the corpus is accessed and searched in a way aligning with the used engine. For example, a relational database storing consecutive n-grams will be efficient when querying a fixed number of tokens. Queries diverging from this structure, however, will perform worse.

2.2 Relational Algebra

The core idea of database engines is to allow usage of a declarative query language to describe the properties of a result set without specifying how results should be fetched. This enables the database engine to decide how a query is executed, by selecting an efficient execution plan. Search and selection of execution plans for a query is guided by a mathematical foundation, a *relational algebra*, as pioneered by IBM's System R (Griffiths Selinger et al., 1979). It describes query operations and defines valid transformations and simplifications. An execution plan is created by selecting one of

[1] https://lucene.apache.org/

the possible query transformations and choosing concrete ways of executing the abstract operations described by relational algebra. This process is guided by a cost model, that incorporates information such as available indexes or size and structure of a table to estimate execution characteristics.

3 Relevance of a Corpus Algebra

We now use the example presented in Figure 1 to demonstrate the usefulness of an algebraic foundation for corpus queries. We do this by recalling the example query and providing different strategies an optimising query engine might use to find a well-performing execution plan. First, let us split the example query up into its three components:

$$A = \texttt{[pos = DET|NUM|PRON]}$$
$$B = \texttt{[pos = ADJ|NOUN]}$$
$$C = \texttt{[deprel = .*(SU|O)BJ]}$$

The original query can be written as $Q = A\,B{*}\,C$.

Each component describes one token out of a token sequence. In an algebraic framework, tokens and results from (sub-)queries can be represented as a set of corpus positions. Sequencing is then modelled using set intersection and disjunction using set union. This enables reasoning about a query's properties in abstract terms.

3.1 Query Planning

One advantage of abstract reasoning is that the query engine has freedom to choose and optimise execution details. The corpus algebra merely defines the properties and nature of the supported operations. In our example, set union and intersection are associative and commutative. This means that the query itself does not specify the order in which results for subqueries are fetched and combined. Instead, the query engine is free to select an execution plan, determining execution order and implementation for all operations in the query.

There are thus many ways to execute a query. To choose the best one, the tool author creates a cost function estimating the execution characteristics of a plan. The cost function encodes knowledge about available implementations for set operations, index lookups and how results are refined. Even the predicted size of results can influence this estimation.

For example, suppose that only two execution plans for our example query existed: Either results for A are fetched from an index and refined, or results for C are fetched from another index and refined. Fetching from an index is very fast, so that the execution time is dominated by the number of results to be refined. A cost function modelling this behavior could use the number of results fetched from an index as the cost of the execution plan. When querying the BNC, around 20.5 million results are fetched for A compared to 24.5 million results for C. Consequently, the first execution plan is better. To handle complex query plans, practical cost functions must be much more complex.

3.2 Query Optimisation Using Algebraic Transformations

Another advantage of abstract reasoning is the ability to find alternative queries describing the same result set. Combined with query planning, a query engine is able to choose from a multitude of execution plans. Algebraic laws provide the foundation to formulate valid transformations of queries.

In our example, the law $B{*} = \varepsilon \mid B\,B{*}$ can be applied. Subsequently reordering the query allows transformation into the following forms:

$$\begin{aligned} Q &= A\,B{*}\,C && (1)\\ &= A\,(\varepsilon \mid B\,B{*})\,C && (2)\\ &= A\,C \mid A\,B\,B{*}\,C && (3)\end{aligned}$$

There is an important difference in how these queries can be executed. To execute query 1, it is difficult to use more than one index (e.g. we can fetch the results of A and filter them to see which match the query). However, to execute 3, we can fetch the results of A, B and C, and use set operations to compute $A\,B \cup A\,C$. This yields fewer result candidates for individual filtering, which speeds up execution.

Alternatively, the related law $\varepsilon \mid B{*}\,B$ could be applied, which leads to transformations (4 – 6). Instead of using the result candidates from A and B for refinement, the query engine can make use of B and C instead. Which variant would be preferred depends on the cost function used.

$$\begin{aligned} Q &= A\,B{*}\,C && (4)\\ &= A\,(\varepsilon \mid B{*}\,B)\,C && (5)\\ &= A\,C \mid A\,B{*}\,B\,C && (6)\end{aligned}$$

In general, applying transformations can both simplify a query and, as in this case, make it more complicated. Making a query more complicated can be worthwhile if, as in this case, the resulting operations are faster to execute – replacing an expensive filter with a cheaper intersection.

3.3 Synergies with other Techniques

The ability to transform queries and find different execution plans for abstract operations also works well in conjunction with other techniques aiming to improve query execution times. By being able to select an execution plan for an abstract query, new optimisation techniques can be added to the planner at any time and are selected whenever applicable. Additionally, queries can be transformed to make more techniques applicable.

As an example, consider binary indexes as presented in Ljunglöf et al. (2024). The transformed queries shown in equations (3) and (6) enable use of a special index to fetch results for a pair of tokens. This way, the entire sub-queries $A\,C$ as well as $A\,B$ and $B\,C$ respectively can be fetched using a single lookup, reducing the number of required operations and further speeding up execution times.

4 Preliminary Results

We have implemented a prototype corpus tool[2] that we described in Ljunglöf et al. (2024). It accepts queries in a subset of the syntax used by Corpus Workbench (CWB, Evert and Hardie, 2011). Currently supported are queries specifying a fixed-length sequence of tokens. Further, our tool supports disjunctions and on the level of single tokens, regular expressions and Boolean conditions. Examples of supported queries can be found in the appendix. Our prototype outperforms CWB on most of the supported queries.

The improved performance results from using multiple indexes per query and optimising the order of operations. Our base assumptions are that index lookups are fast and that performed set operations get faster with smaller operands. The query planner uses these assumptions to execute index lookups for all subqueries with appropriate indexes present and subsequently perform set intersections starting from the smallest sets. It uses algebraic reasoning in two essential ways: firstly, to skip redundant intersections (e.g. intersecting with a unary index A when the binary index $A\,B$ has already been used), and secondly, to tell if the result set can contain false positives. When available indexes do not cover the full query we need to filter results for these false positives afterwards.

As a consequence of this optimising query planning, the execution time for a query depends on the query's size and the *smallest* number of results for any of its subqueries. This contrasts other tools, such as CWB, where the number of results for the *first* subquery defines most of the execution time in addition to the overall query size. In practice this means that many queries will finish almost instantaneously within 100 milliseconds, while the same query might take multiple seconds for CWB to execute. This effect is clearly visible if the first token in the queried sequence is specified in rather broad terms (such as *determiner* or *noun*). The appendix contains an informal evaluation of some queries on CWB compared to our prototype.

The key improvement of an automatically optimising query system is, that a user is free to specify queries whichever way they see fit without tradeoffs in terms of performance. The promising results of our prototype in combination with initial applications of a formal framework, lead us to believe that a corpus algebra is a viable and promising foundation for query optimisation.

5 Planned Contributions

Our goal is to support a much more complex query language, which in turn requires more expressive mathematical foundations and a richer set of transformation laws. To accomplish this, the following four tasks are essential next steps and goals of our research project:

Query language: Extending the query language to enable more expressive queries requires the extension of the mathematical framework. It is important to provide solid mathematical underpinnings for our query language to formally describe the execution behavior of queries.

Corpus algebra: An adequate formal framework is needed to describe the behavior of corpus queries and execution plans. For this purpose, we aim to develop a corpus algebra specialised for the requirements of corpus queries.

Query transformations: Using the corpus algebra, we can describe valid query transformations based on algebraic identities. These transformations do not change the result set of a query.

Query engine: Query language, algebra and transformations provide the foundation to develop a query engine. This engine is capable of automatically optimising, evaluating and selecting fast execution plans for provided queries.

[2] Available from https://github.com/heatherleaf/korpsearch

Acknowledgments

This work has been supported by Språkbanken – jointly funded by its 10 partner institutions and the Swedish Research Council (2018–2024; dnr 2017–00626).

References

J. Bingel and N. Diewald. 2015. KoralQuery – a general corpus query protocol. In *NODALIDA Workshop on Innovative Corpus Query and Visualization Tools*, pages 1–5, Vilnius, Lithuania.

BNC Consortium. 2007. British national corpus, XML edition. Literary and Linguistic Data Service.

M. Davies. 2005. The advantage of using relational databases for large corpora: Speed, advanced queries, and unlimited annotation. *International Journal of Corpus Linguistics*, 10(3):307–334.

N. Diewald and E. Margaretha. 2016. Krill: KorAP search and analysis engine. *Journal for Language Technology and Computational Linguistics*, 31(3):63–80.

S. Evert and A. Hardie. 2011. Twenty-first century Corpus Workbench: Updating a query architecture for the new millennium. In *Corpus Linguistics 2011 Conference*, Birmingham, England.

Sumukh Ghodke and Steven Bird. 2012. Fangorn: A system for querying very large treebanks. In *COLING Demonstration Papers*, pages 175–182, Mumbai, India.

P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and G. T. Price. 1979. Access path selection in a relational database management system. In *SIGMOD'79 international conference on Management of data*, pages 23–34, Boston Massachusetts.

P. Kleiweg and G. van Noord. 2020. AlpinoGraph: A graph-based search engine for flexible and efficient treebank search. In *International Workshop on Treebanks and Linguistic Theories*, pages 151–161, Düsseldorf, Germany.

Thomas Krause and Amir Zeldes. 2016. ANNIS3: A new architecture for generic corpus query and visualization. *Digital Scholarship in the Humanities*, 13(1):118–139.

Peter Ljunglöf, Nicholas Smallbone, Mijo Thoresson, and Victor Salomonsson. 2024. Binary indexes for optimising corpus queries. In *Proceedings of the 20th Conference on Natural Language Processing (KONVENS 2024)*, pages 149–158, Vienna, Austria.

J. Luotolahti, J. Kanerva, and F. Ginter. 2017. Dep_search: Efficient search tool for large dependency parsebanks. In *21st Nordic Conference on Computational Linguistics*, pages 255–258, Gothenburg, Sweden.

U. Manber and G. Myers. 1990. Suffix arrays: a new method for on-line string searches. In *SODA'90, First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, San Francisco, USA.

P. Meurer. 2020. Designing efficient algorithms for querying large corpora. *Bauta: Janne Bondi Johannessen in memoriam, Oslo Studies in Language*, 11(2):283–302.

Jonathan Schaber, Johannes Graën, Daniel McDonald, Igor Mustac, Nikolina Rajovic, Gerold Schneider, and Noah Bubenhofer. 2023. The LiRI corpus platform. In *CLARIN Annual Conference*, pages 145–149, Leuven, Belgium.

Språkbanken Text. 2024. Svenska Wikipedia. [Data set]. Språkbanken Text, Sweden. DOI 10.23695/7DZ6-EK80.

A Informal evaluation

In this appendix we give an informal evaluation of Corpus Workbench (CWB) compared to our prototype tool Korpsearch. We show the execution times of some example queries on two different annotated corpora, the British National Corpus (see table 1), and Swedish Wikipedia (see table 2). Also provided are the number of matches each query produces. The measured execution time includes the total time execution time required by the search tool, including the program startup time. Measurements have been repeated 100 times with the average of all measurements presented below.

All tests were run on the same machine with a 4.4 GHz 12th generation Intel i5-CPU, 16 GiB of working memory and a Phison P0221 NVMe SSD with 2 TiB of storage capacity. The system runs Linux on version 6.11.5. Tests used CWB on version 3.5.0 compiled using version 14.2.1 of the GCC compiler. Our prototype tool *Korpsearch* is run as a Python program using Python version 3.12.7.

		runtime [s]	
Query	N:o matches	*CWB*	*Korpsearch*
`[word="duck"] [pos="VERB"]`	51	0.006	0.05
`[word="duck" & pos!="NOUN"]`	127	0.006	0.06
`[lemma="the"] [pos="ADJ"] [word="duck"]`	26	8.5	0.05
`[lemma="the"] [pos="ADJ"] [word="d.*"]`	53 k	8.6	0.6
`[pos="NOUN"]`	20.0 M	1.1	0.05
`[pos="NOUN"] [pos="NOUN"]`	2.0 M	3.9	0.05
`[pos="NOUN"] [pos="NOUN"] [pos="NOUN"]`	195 k	4.0	0.06

Table 1: Example queries on British National Corpus (BNC Consortium, 2007)

		runtime [s]	
Query	N:o matches	*CWB*	*Korpsearch*
`[word="anka"]`	225	0.002	0.05
`[word="anka"] [pos="VB"]`	17	0.004	0.05
`[pos="JJ"] [word="ankan"]`	5	14.5	0.05
`[pos="JJ"] [word="anka.*"]`	191	14.9	0.06
`[pos="NN"]`	38.5 M	2.1	0.05
`[pos="NN"] [pos="NN"]`	2.5 M	7.0	0.05
`[pos="NN"] [pos="NN"] [pos="NN"]`	240 k	7.2	0.06

Table 2: Example queries on Swedish Wikipedia (Språkbanken Text, 2024)