Evaluating DBMS-based Access Strategies to Very Large Multi-layer Corpora

Roman Schneider
Institut für deutsche Sprache (IDS)
R5 6-13, D-68161 Mannheim /Germany
schneider@ids-mannheim.de

Abstract

Linguistic query systems are special purpose IR applications. As text sizes, annotation layers, and metadata schemes of language corpora grow rapidly, performing complex searches becomes a highly computational expensive task. We evaluate several storage models and indexing variants in two multi-processor/multi-core environments, focusing on prototypical linguistic querying scenarios. Our aim is to reveal modeling and querying tendencies – rather than absolute benchmark results – when using a relational database management system (RDBMS) and MapReduce for natural language corpus retrieval. Based on these findings, we are going to improve our approach for the efficient exploitation of very large corpora, combining advantages of state-of-the-art database systems with decomposition/parallelization strategies. Our reference implementation uses the German DeReKo reference corpus with currently more than 4 billion word forms, various multi-layer linguistic annotations, and several types of text-specific metadata. The proposed strategy is language-independent and adaptable to large-scale multilingual corpora.

Keywords: Very Large Corpora, Multi-layer Annotation, Linguistic Retrieval, Database Management Systems, Concurrency

1. Motivation

In recent years, the quantitative examination of natural language phenomena has become one of the predominant paradigms within (computational) linguistics. Both fundamental research on the basic principles of human language as well as the development of speech and language technology increasingly rely on the empirical verification of assumptions, rules, and theories. More data are mostly seen as better data (Church & Mercer, 1993), and consequently, we notice a growing number of national and international initiatives related to the building of large linguistic datasets for contemporary world languages. Besides written (and sometimes spoken) language samples, these corpora usually contain vast collections of morphosyntactic, phonetic, semantic, etc. annotations, plus text- or corpus-specific metadata.

The downside of this trend is obvious: Complex queries against very large multi-layer corpora (meaning corpora with multiple, potentially concurring annotation layers) quickly become highly computational expensive tasks. So even with specialized applications, our ability to store linguistic data is often bigger than our ability to analyze all this data in detail. In addition, the findings of empirical corpus studies should be traceable and reproducible (Kilgarriff, 2007; Pedersen, 2008).

Much of essential work towards the querying of linguistic corpora goes into data representation, integration of different annotation systems, and the formulation of query languages (Rehm et. al., 2008; Zeldes et. al., 2009; Kepser et. al., 2010; Frick et. al., 2012). We add to this efforts by focusing the scaling problem: As we go beyond corpus sizes of some billion words and at the same time increase the number of possible search keys (linguistically motivated annotations as well as text-specific metadata like publication date, text type, genre, etc.), query costs rise disproportionately. This is due to the fact that unlike traditional IR systems, corpus retrieval systems have to deal not only with the “horizontal” representation of textual data but also with heterogeneous metadata on all levels of linguistic description. And, of course, the exploration of inter-relationships between annotations

becomes more and more challenging as the number of annotation systems increases; e.g., Bański et. al. (2012) discuss the demands and technical issues when developing innovative corpus analysis platforms.

Given this context, we proposed a novel retrieval approach that uses task parallelization and scales well to billion-word corpora (Schneider, 2011). The following sections evaluate speedup effects of this approach in multi-processor/multi-core environments as well as influences of data parallelization (partitioning) and indexing methods.

2. The Reference System

We are using a subset of 4 billion words from the multi-layer annotated German Reference Corpus *DeReKo (Deutsches Referenzkorpus)* (Kupietz et. al., 2010), which constitutes the largest linguistically motivated collection of contemporary German. It covers language data from different media types (literature, newspapers, specialist journals, online texts, etc.), has been annotated morphosyntactically with three competing systems (Connexor, Xerox, TreeTagger), and provides additional text-specific metadata.

Complex data of this kind can be made accessible in different ways. In order to filter out inappropriate approaches, we formulate the following presuppositions:

i. XML/SGML-based markup languages are more suitable for data exchange than for efficient storing and retrieval, so we prefer a compact encoding.

ii. File-based data storage is less robust, flexible, and powerful than the maintenance of text and metadata within database management systems (DBMS).

iii. Although a computer's main memory is still the fastest form of data storage, even with compression techniques it does not seam feasible to rely exclusively on RAM. Attempts to implement (indexless) in-memory databases for considerably large language corpora (Pomikálek et. al., 2009) perform well for unparsed texts but are strongly limited in terms of storage size and, therefore, cannot deal with terabytes of multi-layer annotations.

iv. In order to overcome physical RAM limitations, other approaches use database systems and decompose sequences of strings from the source texts into indexed n-gram tables (Davies, 2005). This results in relatively low query costs and allows fast retrieval with a predefined maximum number of search expressions. However, space requirements for increasing values of n are enormous, and the consuming of system resources for queries spanning more than a handful of words or even sentences, thus, becomes unacceptable. Moreover, complex queries with regular expressions (*Find all tokens that start with a capital letter followed by one or two vocals, and end on 'der'*) or NOT-queries (*Find all tokens that are not classified as nouns*) – both are crucial for comprehensive linguistic exploration – do not benefit from n-gram-based full-string indexes and, thus, perform rather poor.

As a consequence, our corpus storage and retrieval approach uses an object-relational DBMS (64-bit Oracle 11.2.0.1.0 running on CentOS 5.6) with fine-grained data spread across specifically designed tables. Figure 1 shows an excerpt from our conceptual data model as an entity-relationship diagram. Entity types (corpus, text, sentence, word) are displayed as rectangles with their attributes represented as ellipses (computed attributes have dotted borders); relationships are represented as diamond-shape connectors. In order to evaluate speedup and scaling effects, we implemented the entire framework (i.e., data and retrieval procedures) on two independent systems:

i. Single computer, single processor, multi-core: A commodity low-end server driven by a quad-core CPU with 2.67 GHz clock rate and 16GB RAM.

ii. Single computer, multi-processor, multi-core: This symmetric multiprocessing system (SMP) uses eight quad-core microprocessors with 2,3 GHz clock rate and 128GB RAM.

For the reliable measurement of query execution times – and especially to minimize caching effects – we used a "cold" database, meaning that the database instance is opened and that the most relevant caching areas are cleared. Oracle's server-side memory, known as System Global Area (SGA), consists of various components dedicated to different tasks: The dictionary cache holds information about data dictionary objects; the redo log buffer stores uncommitted transactions etc. Most influential for our retrieval runs is probably the buffer cache, i.e., the part of the SGA containing the most recently used data blocks in order to reduce disk I/O. We always cleared it by entering the command ALTER SYSTEM FLUSH BUFFER_CACHE. We intentionally did not flush the Shared Pool (ALTER SYSTEM FLUSH SHARED_POOL) because this part of the SGA stores, among other things, information about user privileges, table structures, etc. as well as optimized execution plans. We believe that especially the latter are essential features of a relational database system and should be taken into account when evaluating its suitability for corpus retrieval - flushing the shared pool seems even more artificial than not flushing it.

3. Query Strategies

Focusing on DBMS-driven corpus storage, the following retrieval strategies look promising:

i. Concatenated SQL joins: This strategy makes use of the relational power of sub-queries and joins and is already used for productive corpus retrieval. Chiarcos et. al. (2008) use an intermediate language between query formulation and database backend; Bird et. al. (2005) present an algorithm for the direct translation of linguistic queries into SQL. This approach uses absolute word positions, and therefore allows proximity queries without limitation of word distances. We implemented an extensible web-based retrieval form (see Figure 2) that can be used to intuitively formulate complex queries using distinct search keys on different metadata types, e.g.: *Find all sentences containing a determiner immediately followed by a proper noun ending on "er", immediately followed by a noun, immediately followed by the lemma "oder", followed by a determiner (any distance), immediately followed by a plural noun, followed by the lemma "sein" (any distance)*. This form invokes a stored procedure that formulates and executes the corresponding SQL query with all necessary joins. The query uses one single token table, corresponding to the "word" entity in Figure 1.

ii. Task separation and parallelization: Programming models like MapReduce support concurrent execution of tasks and, thus, tackle large-data problems. Alhough MapReduce is already in use in a wide range of data-intensive applications (Lin & Dyer, 2010), its principle of "divide and conquer" has not yet been employed for corpus retrieval. We employ an extended MapReduce strategy that regulates the distribution of language data and processor-intensive computation over several CPU cores – or even cluster of machines – and controls the partition of complex linguistic queries into independent processes that can be executed in parallel. Figure 3 illustrates the map/reduce processes for our sample query from above: Within a "map" step, the original query is partitioned into eight separate key-value pairs. Keys represent linguistic units (position, token, lemma, part-of-speech, etc.) values may be the actual content. Again, all queries use the single word table, but they can be processed in parallel and pass their results (sentence/position) to temporary tables. The subsequent "reduce" processes filter out inappropriate results step by step. Usually, multiple reducing steps cannot be executed in parallel because each reduction produces the basis for the next step. But our framework, implemented with the help of PL/SQL stored procedures within the RDBMS, overcomes this restriction by dividing the entire search tree into multiple sub-trees. The reduce processes for different sub-trees are scheduled simultaneously and aggregate their results after they are finished. So the seven reduce steps of our example can be executed quite naturally within only four parallel stages. The parallel framework stores the search results within

the database schema, making it easy to reuse them for further statistical processing. Additional metadata restrictions (e.g., genre, topic, location, date) are translated into separate map processes and reduced/merged in parallel to the main search.

4. Evaluation Scenarios and Results

Executing the concatenated SQL statement (3.i) with eight multi-type search keys against the four-billion word corpus on the single processor system exceeded its capability when applying the joins with traditional B-tree indexes on unpartitioned heap-tables because the nested loops generated an immense workload – we canceled the operation after a runtime of ten hours. The parallel MapReduce search (3.ii) took less than thirty minutes to complete. This strongly indicates that the second approach fits much better for big corpus data and multiple search keys but that further improvements should be carried out. This includes the testing of appropriate index variants that are the key to efficient corpus retrieval (Ghodke & Bird, 2010).

Although we are comparing response times for queries on different server systems and under different settings, our main interest is not to contribute to overall benchmark tests. Database management systems are a widespread and mature technology: Their general advantages and disadvantages have been listed and benchmarked for decades. The most prominent feature that makes them interesting for querying multiple annotated language corpora is probably the flexibility of the (object-)relational approach: Multiple markup layers can be converted into object-relational structures that then can be accessed with the help of SQL; additional metadata such as text type or creation date can be added and queried in the same way. But despite these general advantages, the practical application of database management systems for corpus retrieval is still under-investigated. We aim to reveal tendencies when using RDBMSs and MapReduce for natural language corpus retrieval: Which query strategy seems reasonable under certain conditions? Which storing settings fit to specific language data? Thus, our systematic evaluation concentrates on the following questions:

i. How do SQL joins perform for increasing numbers of search keys? We evaluate this on 1-, 10-, 100-, 1000-, and 4000-million word corpora with rare-, low-, mid-, high-, and top-level search keys. Figure 4 shows the response times in seconds on the single processor server for the query *select count(t1.co_sentenceid) from tb_token t1, (select co_id, co_sentenceid from tb_token where co_token=token1) t3, (select co_id, co_sentenceid from tb_token where co_token = token2) t2 where co_token = token3 and t1.co_sentenceid = t2.co_sentenceid and t1.co_sentenceid = t3.co_sentenceid and t1.co_id < t2.co_id and t2.co_id < t3.co_id*, using three search keys on identical annotation data (token) and single-column indexes that can be queried in parallel. This query simply counts the number of sentences containing three specified tokens (token1, token2, token3) in a fixed order. Compared to similar queries with two search keys (63s for a top-level search, Figure 5) or one search key (6s for a top-level search, Figure 6), the increase of response time on the 4000-million corpus is obviously disproportional (301s). So SQL joins on token data get remarkably less performant for searches with three (or even more) top-frequent search keys, even when making use of in-built query parallelization and the database's cost-based optimizer. On the multi-processor server, the results showed the same tendency: Alhough absolute response times were decreased by a factor of 5 to 6 (e.g., the search for *der* and *die* on the 4000-million corpus completed in 14 seconds instead of 63 seconds, the search for *der*, *die* and *und* completed in 56 seconds instead of 301 seconds), queries still took significantly longer when joining three (or even more) frequent search tokens in a SQL statement.

ii. When sticking to a maximum of two search keys per SQL query (and splitting queries with more search keys to multiple “map” processes

that store their results in temporary tables), how does this solution scale on the multi-processor system? We tested this for three top-frequent tokens (*der/die/und*) on the 4000-million corpus. The search pattern is – from a linguistic point of view – not genuinely interesting, neither semantically nor grammatically. But it provides a perfect test-case for "expensive" corpus requests by initiating the following highly data-intensive SQL statements:

(1) MAP1: *insert into TB_MAP1 (co_sentenceid, co_id) select t1.co_sentenceid, t2.co_id from TB_TOKEN t1, (select co_id, co_sentenceid from TB_TOKEN where co_token='die') t2 where t1.co_token='der' and t1.co_sentenceid = t2.co_sentenceid and t1.co_id < t2.co_id;*

(2) MAP2: *insert into TB_MAP2 (co_sentenceid, co_id) select t1.co_sentenceid, t1.co_id from TB_TOKEN t1 where t1.co_token='und';*

(3) REDUCE: *insert into TB_REDUCE (co_sentenceid) select t1.co_sentenceid from TB_MAP1 t1, TB_MAP2 t2 where t1.co_sentenceid=t2.co_sentenceid and t1.co_id < t2.co_id;*

Statement 1 and statement 2 are scheduled simultaneously; the reduce statement 3 has to wait until both map processes are completed. Figure 7 shows the results on the two servers – please note that – as expected – the insert statements took longer than the queries in 4.i since we are not only counting the number of sentences but also storing the sentence ids in a final result table for further processing. Overall, our tests reveal that the scaling benefit is not strictly linear (32 CPU cores do not perform our MapReduce retrieval eight times faster than 4 cores), but it is promising: The searches on the multi-processor system completed about 4 to 5 times faster than on the single processor system. This corresponds to Amdahl's Law that says that the maximum speedup improvement when using multiple processors is limited by several factors, most prominently by inevitable sequential fractions of the executed tasks. On the other hand, concurrency problems gain weight when parallelization is increased; this phenomenon is addressed by Neil Gunther's Super-Serial Scalability Model. At any rate, our MapReduce approach performed better than the concatenated SQL joins – and the more search keys are used, the difference should be bigger. There may be some potential for further optimization of our mapping algorithm (How should complex queries be divided? How many parallel query tasks are optimal for our system?) as well as for the fine-tuning of database parameters (sizes of memory areas, maximum number of processes, parallelization degrees of tables and indexes, etc.).

iii. How does partitioning of language data improve response times? We partitioned a separate POS table, containing information about each token's word class as identified by the Connexor tagger, according to POS value and sentence number: *create table tb_morpho (co_sentenceid number(10), co_morpho varchar2(10), co_sub varchar2(10), co_id number(10)) partition by range(co_sentenceid) subpartition by list (co_morpho) subpartition template (subpartition A values ('A'), subpartition DET values ('DET'), ... , subpartition PRON values ('PRON'), subpartition P values ('P')) (partition p1 values less than (20000000), partition p2 values less than (40000000), ... , partition p11 values less than (220000000), partition p12 values less than (240000000), partition p13 values less than (MAXVALUE))*. The same partitioning was done for the associated multi-column index. When comparing response times after and before the reorganization, we found that the query s*elect unique t1.co_sentenceid from*

tb_morpho t1, tb_morpho t2 where t1.co_sentenceid = t2.co_sentenceid and t1.co_morpho = 'PRON' and t2.co_morpho='DET' ("find all sentences containing a pronoun and a determiner") on the single processor system was now completed within 50 seconds instead of 300 seconds. So partitioning relational tables holding linguistic data according to often-used search attributes with a small number of distinct values obviously raises their potential for fast query execution. This can be explained with the simple fact that queries for particular POS values no longer have to scan the whole table/index but only certain partitions, and with the possibility to distribute searches on multiple partitions to multiple CPU cores.

iv. How can specific index types improve complex queries? Advanced pattern matching with regular expressions (RegExp) or double/left truncated wildcards is a feature often demanded for linguistic corpus retrieval. Modern database management systems usually contain specific enhancements for their retrieval languages that allow for the integration of regular expressions. For example, Oracle RDBMS offers four RegExp functions that implement the POSIX Extended Regular Expressions (ERE): REGEXP_REPLACE, REGEXP_SUBSTR, REGEXP_LIKE and REGEXP_INSTR. E.g., a SQL search for tokens starting with a capital letter, followed by the substring *'mini'* (at any distance) and ending on the suffix '*er*' (again at any distance) would use the RegExp-enhanced restriction clause *WHERE REGEXP_LIKE (<column name>, '^[[:upper:]].*mini.*er$')*. Unfortunately, standard database index types (b-tree, bitmap) do not support this kind of query because it is impossible to forecast all possible RegExp patterns, so the execution plan will always arrange a time-consuming full table scan. On the other hand, prototypical linguistic searches mostly contain indexable substrings that can be used to speed up the query: Linguists look quite rarely for complex chemical formulas or something like "*a word starting with two numeric digits, followed by 'A' or 'D', followed by a numeric digit between 5 and 8, etc*.". More frequently, they are interested in words ending on a certain suffix or containing a certain stem. In order to improve performance for such queries, Giles (2005) propose to build functional bitmap indexes over all possible substrings of all corpus tokens. These indexes should then be used as primary filters for queries containing regular expressions. Thus, the above query would be superseded by the statement *SELECT unique matchvalue FROM TABLE (getmatches ('tb_token', 'co_token', 'mini')) WHERE REGEXP_LIKE (matchvalue, '^[[:upper:]].*mini.*er$')*, where the in-line table function acts as primary filter by generating a subquery like *select /*+ index_combine(t) */ co_token from tb_token t where (substr(t.co_token, 1,2) = 'mi' and substr(t.co_token, 3,2) = 'ni') or (substr(t.co_token, 2,2) = 'mi' and substr(t.co_token, 4,2) = 'ni') [...]*. Since bitmap indexes can be combined efficiently on the fly, the subquery is expected to complete quite performant. We evaluated this approach and contrasted the results with:

(1) a similar approach that instead of user-built substring indexes uses Oracle's CONTEXT index type (Oracle Corp., 2011). CONTEXT is often used for building text query applications and document classification applications, and provides means to improve left-truncated and double-truncated wildcard searches.

(2) the omission of any primary filter, i.e. the full table scan variant.

Figure 8 addresses the substring filtering and shows the results and execution times (hh:mm:ss) of three sample SQL statements

(without RegExp, but with substring search) that insert the retrieved unique tokens into a temporary table, using the two different primary filters on our single processor server. There is a minor difference regarding the number of results that can be explained by the fact that the manually-built substring indexes cover only the first 100 characters of each token (the corpus contains some outliers with up to 800 characters that we did not index), whereas CONTEXT includes all tokens. Both variants use the four CPU cores in parallel, but the CONTEXT index beats the user-built substring indexes clearly in terms of speed.

Next, we excluded the table inserts as well as the "unique" constraints, and tested some assorted RegExp-enhanced SQL statements that count the occurrences of all matching tokens within our corpus database. The first retrieval run was always without primary filters, then we used the manually-built substring bitmap indexes, and finally the built-in CONTEXT index. Figure 9 displays the corresponding results and response times. They reveal a somewhat unexpected behaviour, since the (indexless) full table scan variant sometimes performed considerably faster than queries with the user-built substring prefiltering. This behaviour did not – at least not always and not significantly – correspond with the amount of data/number of rows that were prefiltered/retrieved. Most likely, several issues generally influence the suitability of the tested approaches for regular expression search: The complexity of the RegExp search pattern, the use of left- or right-side truncation, the language-specific distribution of (initial) word substrings, etc. All these aspects are worth to be investigated in more detail; we see our test runs as a first starting point, outlining the way to an optimal use of database indexing techniques for RegExp retrieval. In any case, however, the best results were again achieved when we deployed the prefiltering restriction with the CONTEXT index. As we see in Figures 10 (full table scan) and 11 (CONTEXT index), the database's explain plan – automatically generated by the Cost Based Optimizer (CBO) with the help of table/index statistics – for this scenario starts with scanning the index, secondly applies the regular expression on the results, and only then accesses the table ("by local index rowid", the most cost-intensive operation of this run).

5. Summary and Outlook

The results of our studies demonstrate that the joining of relational DBMS technology with a parallel computing approach like MapReduce combines the best of both worlds for linguistically motivated corpus retrieval on big datasets. It makes annotated language corpora manageable, eases the reuse and further processing of results, and scales well. As our initial evaluation shows, standard SQL joins do not sufficiently handle queries for complex structures and syntagmatic patterns on very large natural language collections and should be complemented by a dedicated concurrency model. The tests on the multi-processor system demonstrate the suitability of our approach on high-end servers – and further parallelization over multiple machines would most likely benefit even more from the separation of sub-tasks/sub-queries.

Furthermore, we showed that in order to overcome relational bottlenecks, advanced database features like table partitioning and functional indexes can be adapted to the specifics of language corpora. For example, the value-driven partitioning of the POS table significantly improved response times, and introducing custom-tailored substring indexes for the pre-filtering of regular expressions seems to be an effective way to enable advanced pattern matching on big textual data. We will investigate the latter idea in more detail. In addition to the presented retrieval runs, a quick test with flexible combinations of manually-built indexes for word beginnings/endings (to be automatically employed for search expressions with fixed beginnings/endings) and

CONTEXT indexes (for double-truncated RegExp searches) already showed promising results.

For the future, we plan some scheduling refinements of our parallel framework as well as further testing of index types for different types of linguistic data. A comparison with existing corpus retrieval systems (e.g., *Corpus Workbench* or *SketchEngine*) and/or an implementation of our framework on different database management systems would be desirable – not only in terms of response times but also especially regarding flexibility (e.g., how can query results be displayed/processed?) and expandability (e.g., how can additional metadata and concurrent annotation layers be integrated?). In addition, it would be interesting to see how quantitative language laws can be utilized to fine-tune corpus retrieval systems, e.g., how Zipf's law can contribute to the design and filling of token tables.

6. References

Bański, P.; Fischer, P.M.; Frick, E.; Ketzan, E.; Kupietz, M.; Schnober, C.; Schonefeld, O.; Witt, A. (2012). The New IDS Corpus Analysis Platform: Challenges and Prospects. In Proceedings of the Eighth Conference on Language Resources and Evaluation.

Bird, S.; Chen, Y.; Davidson, S.; Lee, H.; Zhen, Y. (2005). Extending XPath to Support Linguistic Queries. In Proceedings of the Workshop on Programming Language Technologies for XML.

Chiarcos, C.; Dipper, S.; Götze, M.; Leser, U.; Lüdeling, A.; Ritz, J.; Stede, M. (2008). A Flexible Framework for Integrating Annotations from Different Tools and Tag Sets. In Traitement Automatique des Langues 49(2), pp. 271-293.

Church, K.; Mercer, R. (1993). Introduction to the Special Issue on Computational Linguistics Using Large Corpora. In Computational Linguistics 19(1), pp. 1-24.

Davies, M. (2005). The advantage of using relational databases for large corpora. In International Journal of Corpus Linguistics 10(3), pp. 307-334.

Frick, E.; Schnober, C.; Bański, P. (2012). Evaluating query languages for a corpus processing system. In Proceedings of the Eighth International Conference on Language Resources and Evaluation.

Giles, D. (2005). Oracle bitmap Indexes and their use in pattern matching. Unpublished Technical Whitepaper. *http://dominicgiles.com/technicalpapers.html*

Ghodke, S.; Bird, S. (2010). Fast query for large treebanks. In Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics, pp. 267-275.

Kepser, S.; Mönnich, U.; Morawietz, F. (2010). Regular Query Techniques for XML-Documents. In Metzing, D.; Witt, A. (Eds.): Linguistic Modeling of Information and Markup Languages, Springer, pp. 249-266.

Kilgarriff, A. (2007). Googleology is bad science. In Computational Linguistics, 33(1), pp. 147-151.

Kupietz, M.; Belica, C.; Keibel, H.; Witt, A. (2010). The German Reference Corpus DeReKo: A Primordial Sample for Linguistic Research. In Proceedings of the Seventh Conference on Language Resources and Evaluation (LREC 2010), pp. 1848-1854.

Lin, J.; Dyer, C. (2010). Data-Intensive Text Processing with MapReduce. Synthesis Lectures on Human Language Technologies. Morgan & Claypool.

Oracle Corp. (2011). Text Application Developer's Guide 11*g* Release 2 (11.2). *http://docs.oracle.com/cd/E11882_01/text.112/e24435.pdf*

Pedersen, T. (2008). Empiricism Is Not a Matter of Faith. In Computational Linguistics, 34(3), pp. 465-470.

Pomikálek, J.; Rychlý, P.; Kilgarriff, A. (2009). Scaling to Billion-plus Word Corpora. In Advances in Computational Linguistics 41, pp. 3-13.

Rehm, G.; Schonefeld, O.; Witt, A.; Chiarcos, C.; Lehmberg, T. (2008). A Web-Platform for Preserving, Exploring, Visualising and Querying Linguistic Corpora and Other Resources. In Procesamiento del Lenguaje Natural 41, pp. 155-162.

Schneider, R. (2011). A functional database framework for querying very large multi-layer corpora. In Proceedings of the GSCL Conference 2011.

Zeldes, A.; Ritz, J.; Lüdeling, A.; Chiarcos, C. (2009). ANNIS: A Search Tool for Multi-Layer Annotated Corpora. In Proceedings of Corpus Linguistics 2009.

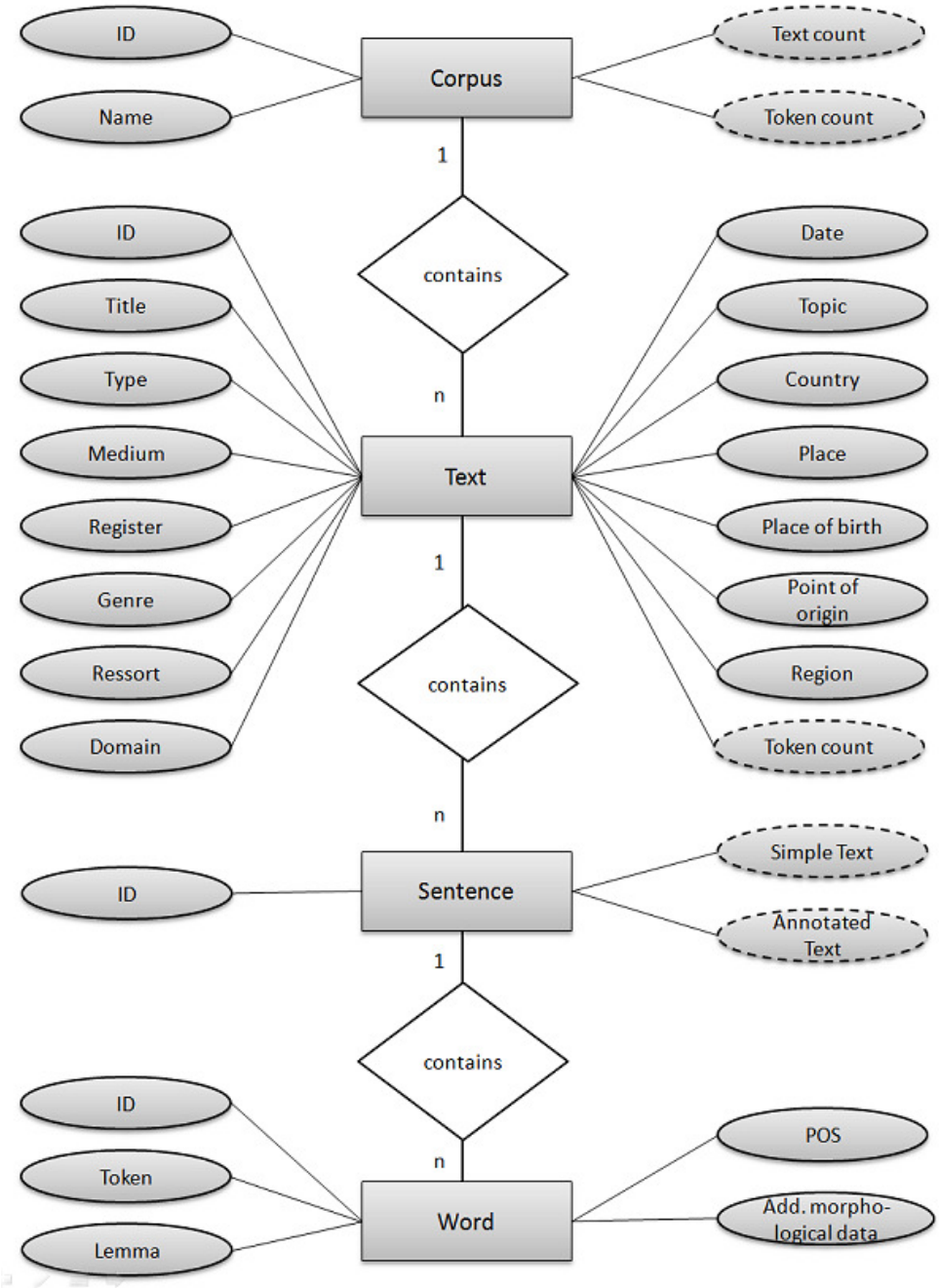

Figure 1: Semantic/conceptual data model (excerpt).

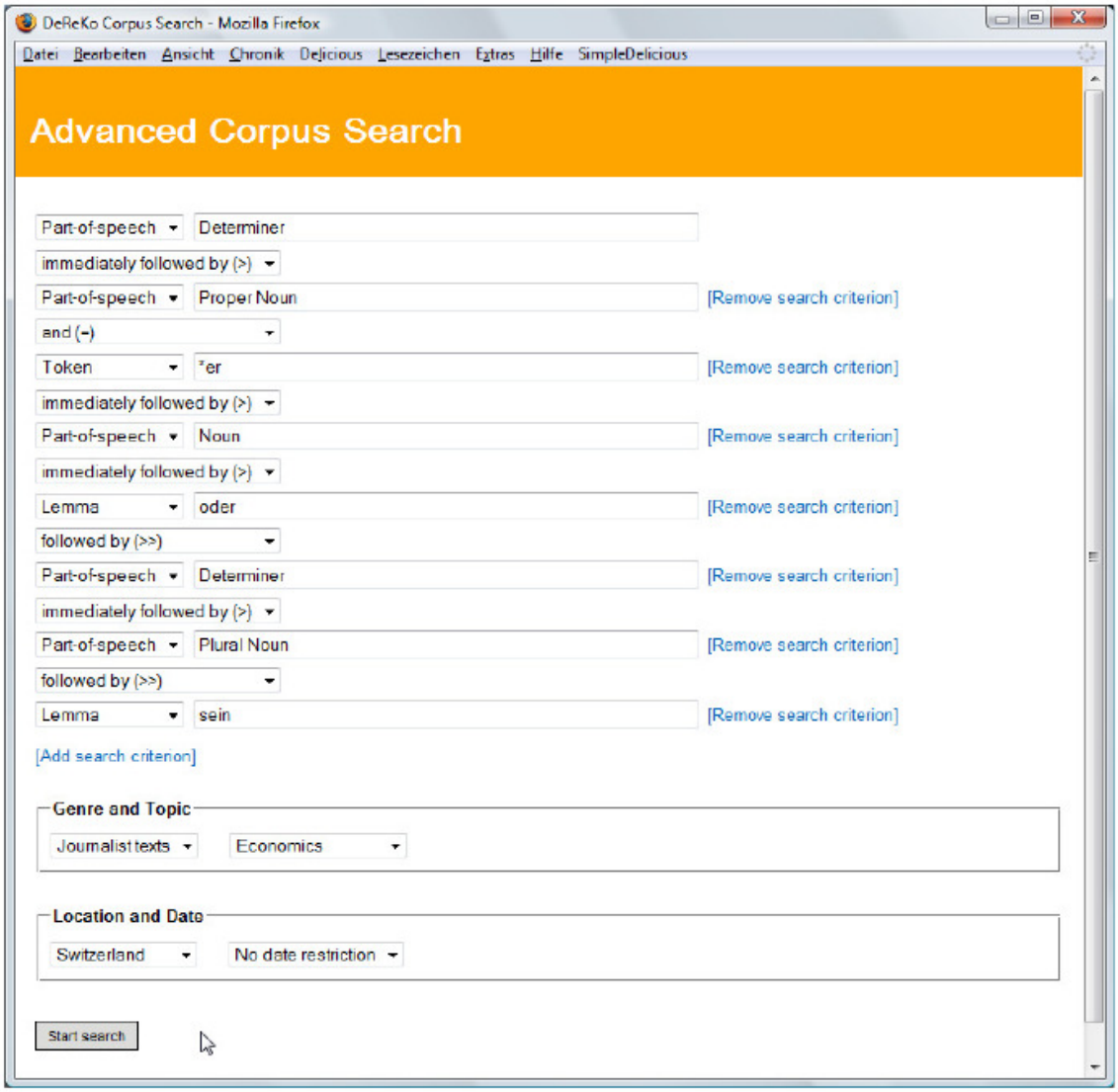

Figure 2: Web-based retrieval form with our sample query.

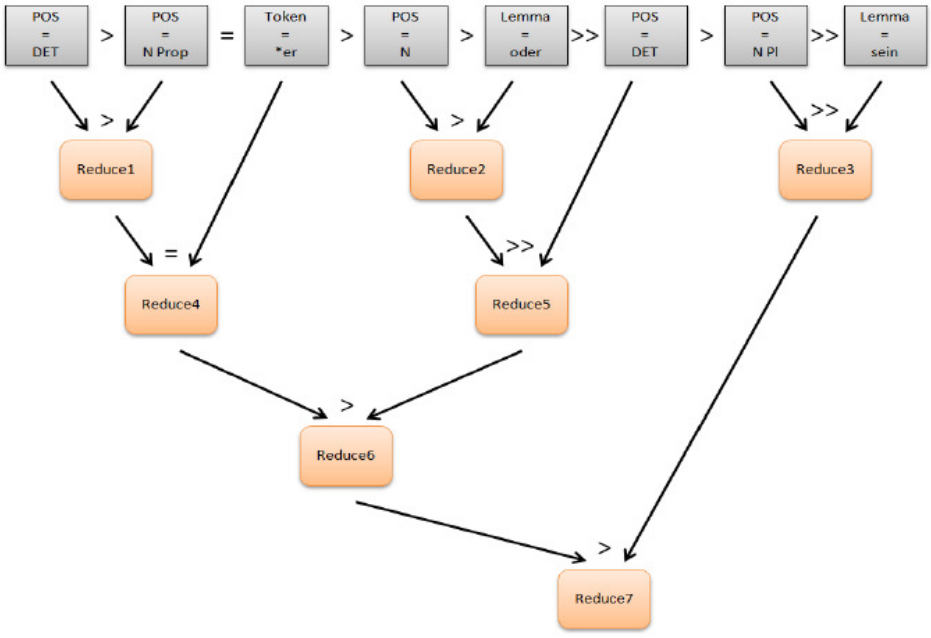

Figure 3: MapReduce processes for a concatenated query with eight search keys.

	1 Mio	10 Mio	100 Mio	1000 Mio	4000 Mio
rare (*traurige/isoliert/trauen*)	0,03	0,16	0,3	0,6	0,9
low (*langsam/lesen/verfügt*)	0,29	0,35	0,37	0,65	1
mid (*ohne/nun/uns*)	0,3	0,4	0,5	1,2	3,8
high (*nicht/ist/dem*)	0,31	0,47	1,49	10,47	38,7
top (*der/die/und*)	0,4	0,87	4,67	47,06	301

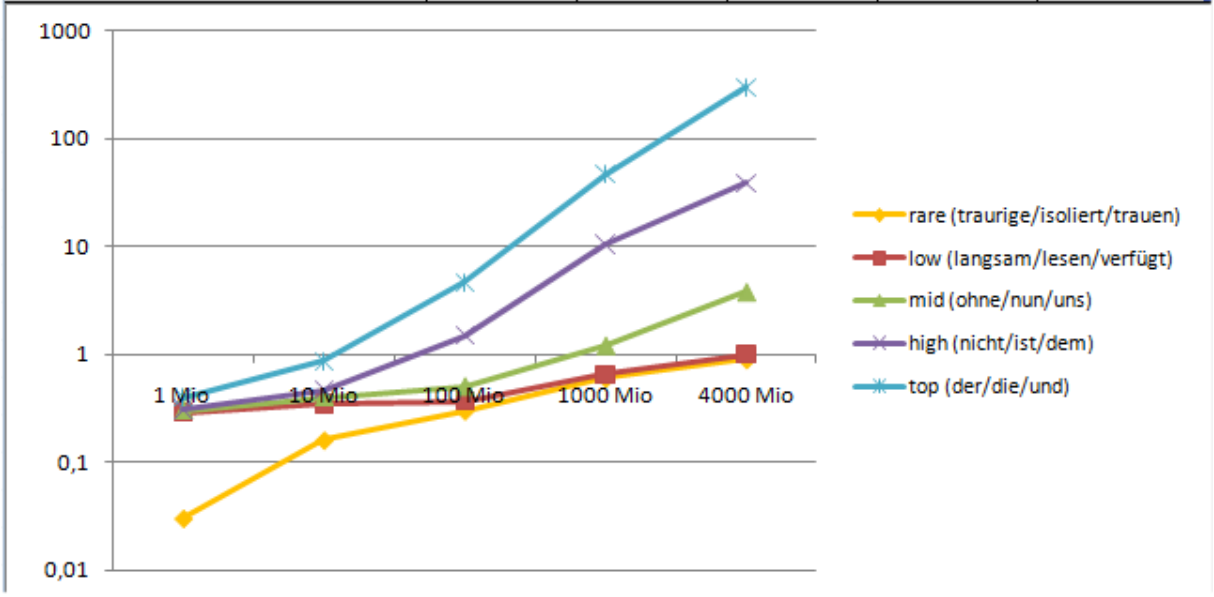

Figure 4: Response times (s) for nested SQL queries with three search keys (logarithmic scaled axis).

	1 Mio	10 Mio	100 Mio	1000 Mio	4000 Mio
rare (*traurige/isoliert*)	0,16	0,19	0,29	0,3	0,5
low (*langsam/lesen*)	0,09	0,2	0,3	0,4	1,3
mid (*ohne/nun*)	0,16	0,23	0,31	0,9	2,3
high (*nicht/ist*)	0,21	0,32	0,89	6,8	12,7
top (*der/die*)	0,32	0,65	3,12	56	63,3

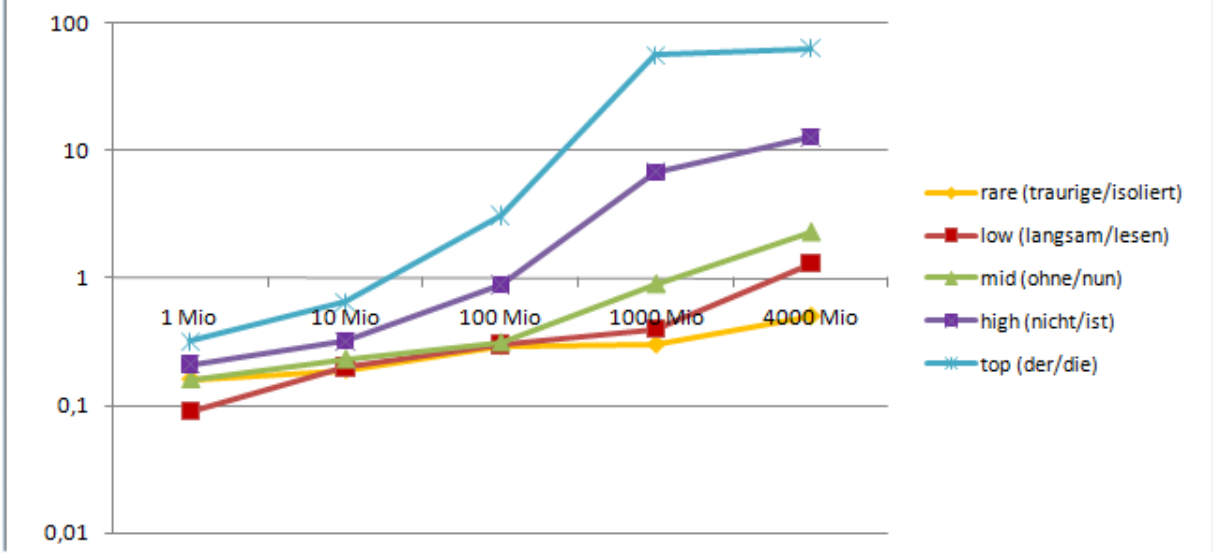

Figure 5: Response times (s) for nested SQL queries with two search keys (logarithmic scaled axis).

	1 Mio	10 Mio	100 Mio	1000 Mio	4000 Mio
rare (*traurige*)	0,04	0,05	0,1	0,2	0,2
low (*langsam*)	0,03	0,05	0,2	0,3	0,3
mid (*ohne*)	0,1	0,1	0,2	0,3	0,3
high (*nicht*)	0,21	0,28	0,45	1	1
top (*der*)	0,29	0,49	0,92	5	6

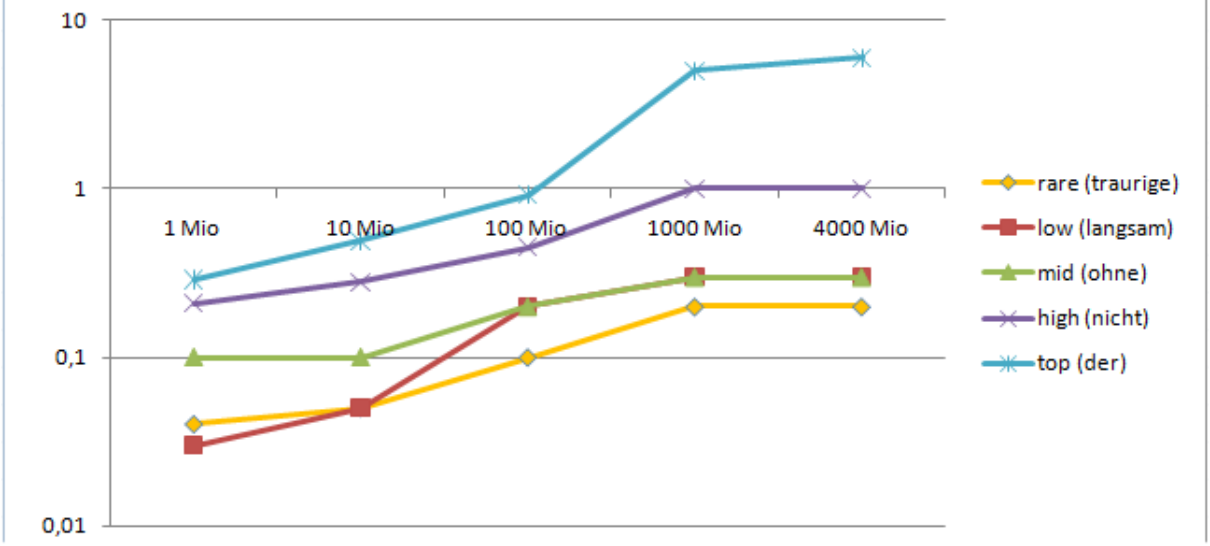

Figure 6: Response times (s) for nested SQL queries with one search key (logarithmic scaled axis).

	Single CPU Server	Multi CPU Server
(1) MAP1	112	28
(2) MAP2	18	5
(3) REDUCE	46	9
Total	176	42

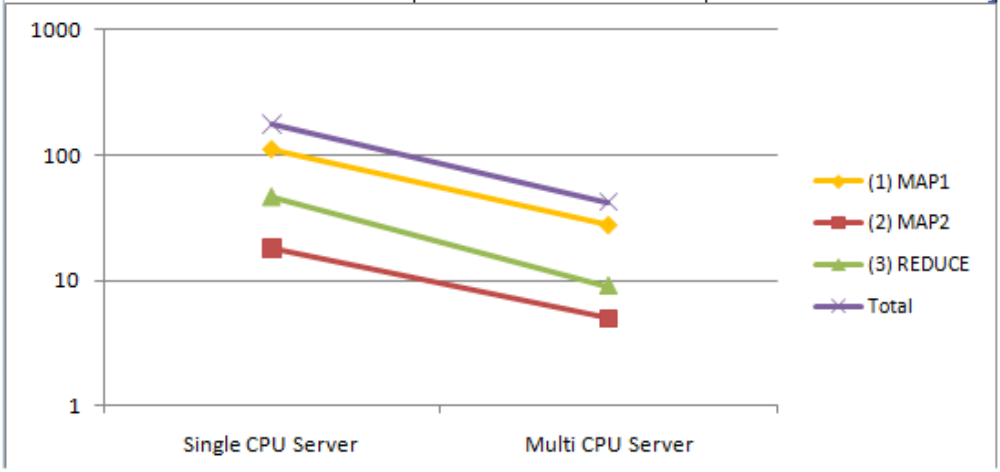

Figure 7: Execution times (s) for MapReduce processes on the 4-billion word corpus with three top frequent search keys (logarithmic scaled axis).

		Substring Indexes	Context Index
SQ	SQL statement [1]	insert into tb_map select unique matchvalue from table(getmatches('tb_connexor_word', 'co_token', 'haupt'));	insert into tb_map select unique co_token from tb_connexor_token where contains (co_token, '%haupt%')>0;
Nu	Number of results [1]	10130	10136
Sa	Sample results [1]	Geistesoberhaupt Forschungshauptleiter Gebäudehauptverteilung Paprikahauptstadt Oberhauptstimme "Wachstumshauptstadt"	Landeshauptmannstellverrtreter Luftwaffenhauptamtes "Schreckenshaupt" Eifelhauptvereins www.tu-berlin.de/zuv/asb/aktuell/haupt.html Inselhauptmotiv
Ex	Execution time [1]	00:22:56	00:00:18
SQ	SQL statement [2]	insert into tb_map select unique matchvalue from table(getmatches('tb_connexor_word', 'co_token', 'mini'));	insert into tb_map select unique co_token from tb_connexor_token where contains (co_token, '%mini%')>0;
Nu	Number of results [2]	34603	34610
Sa	Sample results [2]	Exfamilienministerin Femininen Feministenkreisen Forschungsministerium Postministern	Chefadministrators Eliminierungstaktik Fachministerinnen Exgesundheitsministerin ministerpräsidentiell
Ex	Execution time [2]	01:47:20	00:02:39
SQ	SQL statement [3]	insert into tb_map select unique matchvalue from table(getmatches('tb_connexor_word', 'co_token', 'http://www.'));	insert into tb_map select unique co_token from tb_connexor_token where contains (co_token, '%http://www.%')>0;
Nu	Number of results [3]	59523	59529
Sa	Sample results [3]	http://www.lzg-rlp.de "http://www.kfa-juelich.de/compchem"	http://www.texterschule.de Unterhttp://www.snowcrest.net/mindport/agent.html
Ex	Execution time [3]	00:02:57	00:00:01

Figure 8: Results and execution times (hh:mm:ss) for sample queries using the two different prefilter indexes.

	Full Table Scan	Substring Indexes	Context Index
SQL statement [1]	SELECT count(*) from tb_connexor_token WHERE REGEXP_LIKE (co_token, '^[[:upper:]].*mini.*er$');	SELECT count(*) FROM TABLE (getmatches ('tb_connexor_word', 'co_token', 'mini')) WHERE REGEXP_LIKE (matchvalue, '^[[:upper:]].*mini.*er$');	SELECT count(*) from tb_connexor_token WHERE contains (co_token,'%mini%')>0 and REGEXP_LIKE (co_token, '^[[:upper:]].*mini.*er$);
Number of results [1]	1251180	*see Full Table Scan*	*see Full Table Scan*
Sample results [1]	Schattenwirtschaftminister Aluminiumschweißer Dominialgüter	*see Full Table Scan*	*see Full Table Scan*
Execution time [1]	00:27:43	01:49:21	00:15:39
SQL statement [2]	SELECT count(*) from tb_connexor_token WHERE REGEXP_LIKE (co_token, '^.*haupt.*ung$');	SELECT count(*) FROM TABLE (getmatches ('tb_connexor_word', 'co_token', 'haupt')) WHERE REGEXP_LIKE (matchvalue, '^.*haupt.*ung$');	SELECT count(*) from tb_connexor_token WHERE contains (co_token,'%haupt%')>0 and REGEXP_LIKE (co_token, '^.*haupt.*ung$');
Number of results [2]	147133	*see Full Table Scan*	*see Full Table Scan*
Sample results [2]	Mütterhauptversammlung Radiohauptabteilung Rettungshundehauptprüfung	*see Full Table Scan*	*see Full Table Scan*
Execution time [2]	00:36:07	00:22:56	00:12:01
SQL statement [3]	SELECT count(*) from tb_connexor_token WHERE REGEXP_LIKE (co_token, '^http://www\..*\.[[:alpha:]]{3}$');	SELECT count(*) FROM TABLE (getmatches ('tb_connexor_word', 'co_token', 'http://www.')) WHERE REGEXP_LIKE (matchvalue, '^http://www\..*\.[[:alpha:]]{3}$');	SELECT count(*) from tb_connexor_token WHERE contains (co_token,'http://www.%')>0 and REGEXP_LIKE (co_token, '^http://www\..*\.[[:alpha:]]{3}$');
Number of results [3]	16069	*see Full Table Scan*	*see Full Table Scan*
Sample results [3]	http://www.mikrocontroller.net http://www.goto.com http://www.ilslaunch.com/index.cgi	*see Full Table Scan*	*see Full Table Scan*
Execution time [3]	00:31:41	00:04:00	00:00:10
SQL statement [4]	SELECT count(*) from tb_connexor_token WHERE REGEXP_LIKE (co_token, '^[[:lower:]].*bar$');	SELECT count(*) FROM TABLE (getmatches ('tb_connexor_word', 'co_token', 'bar')) WHERE REGEXP_LIKE (matchvalue, '^[[:lower:]].*bar$');	SELECT count(*) from tb_connexor_token WHERE contains (co_token,'%bar')>0 and REGEXP_LIKE (co_token, '^[[:lower:]].*bar$');
Number of results [4]	2156987	*see Full Table Scan*	*see Full Table Scan*
Sample results [4]	aufsplittbar normalbefahrbar beskatebar	*see Full Table Scan*	*see Full Table Scan*
Execution time [4]	00:27:36	00:14:56	00:09:56
SQL statement [5]	SELECT count(*) from tb_connexor_token WHERE REGEXP_LIKE (co_token, '^Ober[[:alpha:]]{5}er$');	SELECT count(*) FROM TABLE (getmatches ('tb_connexor_word', 'co_token', 'Ober')) WHERE REGEXP_LIKE (matchvalue, '^Ober[[:alpha:]]{5}er$');	SELECT count(*) from tb_connexor_token WHERE contains (co_token,'Ober%')>0 and REGEXP_LIKE (co_token, '^Ober[[:alpha:]]{5}er$');
Number of results [5]	53762	*see Full Table Scan*	*see Full Table Scan*
Sample results [5]	Oberpfeifer Oberbergler Oberwandter	*see Full Table Scan*	*see Full Table Scan*
Execution time [5]	00:31:55	01:49:18	00:00:35

Figure 9: Results and execution times (hh:mm:ss) for sample RegExp queries with and without prefilter indexes.

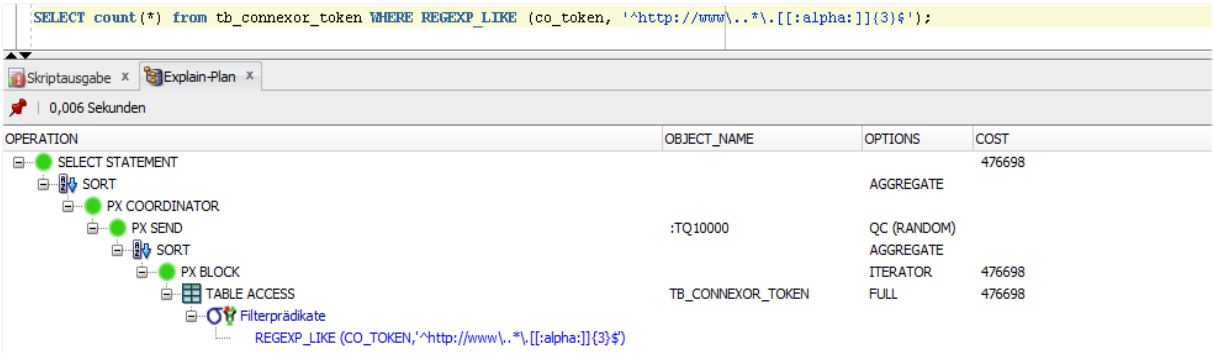

Figure 10: Explain plan for sample RegExp queries without prefiltering.

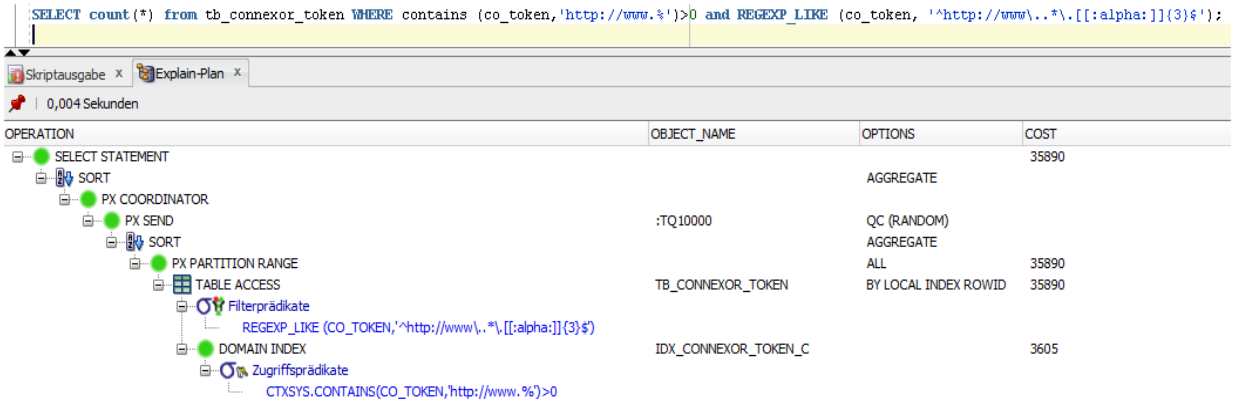

Figure 11: Explain plan for sample RegExp queries with CONTEXT-based prefiltering.