Developing Corpus Management System: Architecture of System and Database

O. Nevzorova\textsuperscript{1,2}, D. Mukhamedshin\textsuperscript{2}, and R. Gataullin\textsuperscript{2}

\textsuperscript{1}Kazan Federal University, Kazan, Russia
\textsuperscript{2}Institute of Applied Semiotics of the Tatarstan Academy of Sciences, Kazan, Russia

Abstract - The paper describes two basic stages of corpus management search engine development: the system architecture development and the database architecture development. The proposed solutions have been implemented in the corpus management system which is currently used for data representation and processing for the National Corpus of Tatar “Tugan Tel”.

Keywords: corpus data, database, software architecture, highload applications, search engine.

1 Introduction

Developing corpus management search engine architecture and database implies analyzing all the necessary functionality and testing various DBMS. Correct, fast enough and optimal in terms of resource consumption search engine performance depends entirely on a proper system and database architecture.

The development of the Tatar corpus management system (www.corpus.antat.ru) was aimed at preservation and progression of the Tatar language through new information technologies, providing advanced search capabilities for the database of the Tatar Language Corpus [10]. Researches into the Tatar language heavily depend on the use of the Tatar corpus management system in humanitarian and educational applications, as well as in applications belonging to computational linguistics sphere.

Ready-made solutions are often used for developing national corpuses, which has its pros and cons. In particular, such solutions are usually proprietary. Some of them will be discussed in Section II.

2 Related works

There are a lot of works devoted to the development of corpus management systems.

Many search engines which work with national linguistic corpuses use ready kernels such as Yandex.Server, for instance the Russian National Corpus [11] search engine. Such systems represent a complex of several subsystems, which provides a fast and multifunctional search. Yandex.Server is proprietary and its full version is distributed on a commercial basis. The system enables to execute direct and reverse search queries, to use logical expressions in morphological queries (AND, OR operators), but it has no speed optimization for morphological search since it was developed as a system for direct search and search based on Russian morphology (on lemmas). The use of such systems requires fine-tuning of hardware and software environment to achieve the best results.

Another important work is the Czech National Corpus [https://www.korpus.cz/] [7]. It is based on the Sketch Engine [3, 6] system which has the following advantages [4, 8, 9]: it supports arbitrary metadata of documents, uses its own query language (CQL – Corpus Query Language), supports reverse search and phrase search, allows one to view corpus statistics and to select lists of words and n-grams by parameters; it also handles documents in different formats and uses NoSQL database [5]. But the Sketch Engine has some limitations too. Among these are the following: it does not show the morphological markup, has complicated reverse search, processes search queries for a long time, has no speed optimizations, demonstrates DDoS vulnerability, and has an extensive search engine results page.

3 System architecture

As its software implementation, the Tatar Corpus management system uses the MVC (Model-View-Controller) concept which is slightly modified to solve the system problems.

Figure 1. Abstract scheme of task processing in the system.
The general scheme of any task entrusted to the system processing is shown in Figure 1.

A more detailed scheme of the system elements and interactions between them (the system architecture) is shown on the map (Figure 2). It displays the relations between all elements of the system. The system operation begins with the Request which enters the Main Controller (MainControl). First of all, the Main Controller provides security using the Shield component (ShieldModel) which in turn uses the Queue object. If the Shield considers the Request to be safe, further control is passed to a specific controller, depending on the action required. There are 7 specific controllers in the system: Single Page Control, Search Control, Context Control, Single Page Edit Control, Statistics Control, Data Management Control, and Security Control. The first three use the abstract page controller PageControl and have public access. PageControl combines the general data control functionality for pages with public access. The remaining are management controllers that use the AdminControl abstract management pages’ controller and are available to users with management privileges (administrators and editors). AdminControl combines the general functionality of management pages’ controllers and inherits PageControl functionality. PageControl and AdminControl use the auxiliary requests controller RequestControl for easy data check of user queries.

After checking and filtering the data, the system passes control to a model that corresponds to the requested action: Single Page Model, Search Model, Query Model (is not called directly from the controller and is used by the Search Model), Context Model, Single Page Edit Model, Statistics Model, Data Management Model (uses Document Model, Sentence Model and Word Model that are not displayed on the map), or Security Model. All models inherit the PageModel that combines pages functionality. The last four models use management page model AdminPageModel that combines functionality for management pages. All models use the database model DB, which in turn employs the Cache model.

Once the model action is executed, the control returns to the controller from where the data are transmitted to the View (View). The latter uses the data and the corresponding page templates to generate an HTML-document, or returns the data in JSON, depending on the requested output format.

4 Database and data storage choice
4.1 Database and data storage search and analysis

In the first stage of system development it was necessary to choose the DBMS and the data storage to be used in the search engine. The potential DBMS and data storages had to be designed to achieve a fast and reliable access to a large amount of data in real time. Those had to match the following criteria:

- Performance (search speed through the database including 100 million rows in the table – at least 1 query per second);
- Scalability (compliance with the requirements of the functionally expanded system, with the distribution of processes to multiple physical machines);
- Price (the analysis considered free and commercial DBMS);
- Compatibility with software (the ability to work with the PHP and Unix-like OS support);
- Availability of documentation (availability of detailed documents in Russian or English);
- Perspectives (dynamics of the project development, existing users community, plans of developers).

We have searched and analyzed DBMS and data storages matching these criteria based on the information from publicly available sources. 11 potential DBMS and data storages were selected:

1. Memcached/MemcacheDB (http://memcached.org/, http://memcachedb.org/);
2. Redis (http://redis.io/);
3. Tarantool (http://tarantool.org/);
4. Aerospike (http://www.aerospike.com/);
5. FoundationDB (https://foundationdb.com/);
6. Apache HBase (http://hbase.apache.org/);
7. Hypertable (http://hypertable.com/);
8. Cassandra (http://planetcassandra.org/);
9. MariaDB (https://mariadb.org/);
10. Sphinx (http://sphinxsearch.com/);

Inverse index storage structures have been developed for each candidate.

4.2 Performance test of databases and data storages

Each system was installed on the virtual test machine, with further performance testing. The virtual machine had the following technical characteristics: 4 cores CPU (2.7GHz per each), 4GB DDR3 RAM, HDD 20GB (5400RPM), SSD 1GB for swap. The virtual machine was running Debian 7.5. The writing and reading speed were tested for each storage system. The written data (the reverse index of the text) were generated by randomly selecting words from the dictionary, with morphological properties randomly generated for each word.
Special software has been developed by the authors to generate, write and read the data, as well as to measure performance time for each storage system. This software allowed us to automate the testing and to provide the necessary load on the system. During the performance testing we obtained the results shown in Table 1.

<table>
<thead>
<tr>
<th>System</th>
<th>Writing, words/sec.</th>
<th>Reading, words/sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memcached</td>
<td>3283</td>
<td>13,2</td>
</tr>
<tr>
<td>MemcacheDB</td>
<td>560,7</td>
<td>7,8</td>
</tr>
<tr>
<td>Redis</td>
<td>3792</td>
<td>14</td>
</tr>
<tr>
<td>FoundationDB</td>
<td>692,2</td>
<td>5,8</td>
</tr>
<tr>
<td>Hypertable</td>
<td>2655</td>
<td>15,8</td>
</tr>
<tr>
<td>MySQL (MariaDB)</td>
<td>1031,5</td>
<td>9,9</td>
</tr>
<tr>
<td>Redis+MySQL (MariaDB)</td>
<td>551,3</td>
<td>14,1</td>
</tr>
<tr>
<td>Sphinx</td>
<td>1368,7</td>
<td>0,2</td>
</tr>
<tr>
<td>ElasticSearch</td>
<td>3546,7</td>
<td>11,4</td>
</tr>
</tbody>
</table>

Hypertable demonstrated the highest speed. However, the combination of Redis + MySQL was chosen as the system for further work, since for a small loss of speed, it would significantly extend the search functionality just by changing the query to the database.

5 Database architecture

5.1 Search query types and their features

The system and database architecture was constructed to be able to respond to the following types of queries:

- Direct search (by word form or lemma);
- Reverse search (by morphological properties which can be represented as a formula of:
  - Conjunction;
  - Disjunction;
  - Negation;
  - Arbitrary)
- Hybrid search (by morphological properties and word form or lemma).

The marked text is parsed into individual word forms and their markup to be further written into the database as an inverse index, a list of sentences (contexts) and a list of documents. The search is performed on the reverse index, which significantly increases the speed of executing queries to the database.
As morphological properties are written into the database in the form of binary vectors, the search queries on the conjunction can be represented as superposition of the bit mask of the morphological properties vector from the query over the morphological properties vector from the inverse index table. The result of this operation should equal to the morphological properties vector from the query. Queries on the disjunction are built likewise, but the result of the operation should contain any non-empty vector (at least one 1 must occur). When requesting on the negation the superposition of the bit mask should give the empty vector (all zeros). Superposition of bit masks is a low-cost operation, and although the indexes on the table are not applied to it, queries are executed in less than 1 second.

5.2 Inverse index table

The general structure of the table of inverse index on search engine data is shown in Table 2.

Table 2. Structure of inverse index table.

<table>
<thead>
<tr>
<th>Row name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>INT (AUTO-INCREMENT)</td>
<td>The unique identifier of morphological analysis.</td>
</tr>
<tr>
<td>word</td>
<td>MEDIUMINT UNSIGNED</td>
<td>The unique identifier of a specific word form. Relations between identifiers and word forms are stored in Redis.</td>
</tr>
<tr>
<td>lemma</td>
<td>MEDIUMINT UNSIGNED</td>
<td>The unique identifier of a specific lemma. Relations between identifiers and lemmas are stored in Redis.</td>
</tr>
<tr>
<td>doc</td>
<td>MEDIUMINT UNSIGNED</td>
<td>The unique identifier of a document. Documents are stored in a separate table.</td>
</tr>
<tr>
<td>pos</td>
<td>MEDIUMINT UNSIGNED</td>
<td>Word position in the document.</td>
</tr>
<tr>
<td>sentence</td>
<td>INT UNSIGNED</td>
<td>Sentence number in the document. Sentences are stored in a separate table.</td>
</tr>
<tr>
<td>sentence_pos</td>
<td>SMALLINT UNSIGNED</td>
<td>Word position in the sentence.</td>
</tr>
<tr>
<td>morph1</td>
<td>BIGINT UNSIGNED</td>
<td>The first 64 of the morphological properties.</td>
</tr>
<tr>
<td>morph2</td>
<td>BIGINT UNSIGNED</td>
<td>The following 64 (up to 64) of the morphological properties.</td>
</tr>
</tbody>
</table>

The inverse index table has also several indexes to optimize queries. These indexes are shown in Table 3.

Table 3. Indexes in the inverse index table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type and components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIMARY</td>
<td>PRIMARY KEY (id, word, lemma, morph1)</td>
<td>The composite primary key for a quick search for a specific word, and for organizing partitioning.</td>
</tr>
<tr>
<td>docs</td>
<td>KEY (doc)</td>
<td>Index of document identifiers for a quick search within the document and for grouping through the documents.</td>
</tr>
<tr>
<td>sentences</td>
<td>KEY (sentence)</td>
<td>Index of sentence identifiers for a quick grouping through the sentences and for multiple queries search within the same sentence</td>
</tr>
<tr>
<td>context</td>
<td>KEY (id, doc, sentence)</td>
<td>A composite index for expanding contexts.</td>
</tr>
<tr>
<td>word</td>
<td>KEY (word)</td>
<td>Index of word forms identifiers for a quick direct and hybrid search by wordform.</td>
</tr>
<tr>
<td>lemma</td>
<td>KEY (lemma)</td>
<td>Index of lemma identifiers for a quick direct and hybrid search by lemma.</td>
</tr>
</tbody>
</table>

Moreover, for the inverse index table we created sections and subsections [2], which significantly increased the speed of query executing. The first block in the morphological properties vector (class [1]) determines to which section the specific parsing should be written and the identifiers of word forms and lemmas distribute the row to the subsection.

Sections p1-p14 are defined based on morphological property from the first block (class); “Class_not_found” includes all parsings which do not have the property in the first block; section p0 includes word forms without parsing; punctuation marks at the end of the sentence (points, exclamation marks, question marks) are stored in “End” section; “Syntax” stores all the remaining punctuation; single letter words are stored in “Letter” section; “Latin” stores all the words in which there are Latin characters; all the remaining entries containing special characters are stored in “Sign” section.

Thus, a table of 12.5 GB in size was divided into smaller tables, each of which might be placed in RAM for a quick access to data. This greatly increased the speed of query
executing. With the accumulation of the querying history, sections begin to fill the RAM which stores only the most demanded of them.

6 Conclusion

The system architecture and database architecture discussed in this paper are implemented in the search module of linguistic data management system operating with an electronic corpus of the Tatar language.

In order to revise its adequacy, accuracy, consistency and the character of temporal changes we built a comprehensive testing system. Adequacy tests revealed that the proposed method completely solved the tasks. Accuracy tests based on comparison with other reference methods of representation and processing of queries showed that the methods proposed worked properly.

Testing consistency and the character of changes in time demonstrated that the proposed syntax of lexical and morphological components of a query system was interpreted correctly, and the time required for processing and executing a search query was not more than 0.05 seconds in 98.71% of cases for lexical search and 98.08% of cases for combined lexical-morphological search.

The use of the proposed architecture for linguistic corpuses allows one to address a wide range of problems. This paper describes a few of them. In the future it is planned to expand this architecture to integrate various services related to the processing of linguistic data such as morphological analyzer, morphological ambiguity resolving module, and others.

The described approach to solving search engine problems of linguistic corpuses might allow researchers to use the developed system not only with the electronic corpus of texts in the Tatar language, but also with the corpuses of other languages without significant changes made to the system.

7 Acknowledgment

The reported study was funded by Russian Science Foundation (research project № 16-18-02074).

8 References


