Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «**Информатика и системы управления**»

КАФЕДРА «**Программное обеспечение ЭВМ и информационные технологии**»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Разработка классического статического сервера для отдачи контента с диска»

Студент группы **ИУ7-75Б** _____ **П. А. Иванов**
(Подпись, дата) (И.О. Фамилия)

Руководитель курсовой работы _____ **Н.О. Яковидис**
(Подпись, дата) (И.О. Фамилия)

***2023** г.*

РЕФЕРАТ

Расчетно-пояснительная записка 59 с., 19 рис., 2 табл., 44 ист., 2 прил.

ПАРАЛЛЕЛЬНЫЙ КОРПУС, РАЗМЕТКА ПАРАЛЛЕЛЬНОГО КОРПУСА, БАЗЫ ДАННЫХ, РЕЛЯЦИОННЫЕ СУБД, КЕШИРОВАНИЕ, ЛОГИРОВАНИЕ

Объектом исследования данной работы является корпус параллельных текстов.

Целью работы является разработка базы данных для хранения и обработки параллельного корпуса переведённых текстов.

Поставленная цель достигается путем анализа предметной области параллельных корпусов текстов и существующих систем управления базами данных, проектированием и реализацией базы данных для хранения и пополнения параллельного корпуса. Кроме того, реализована система кеширования и исследовано влияние её использования на время исполнения запросов к системе.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

API – Application Programming Interface [1];

ВВЕДЕНИЕ

Современные сервера должны уметь обрабатывать запросы многих пользователей одновременно. Как правило, для этого применяют три основные техники: мультиплексирование, создание новых процессов (англ. forking) и создание новых потоков (англ. threading). Существуют также модификации последних: pre-forking и pre-threading, идея которых заключается в уменьшении времени задержки ответа путем использования пула процессов или потоков соответственно [2]. Такие сервера, как правило, работают на сокетах – абстракции конечных точек соединения, для работы с которыми в Unix предусмотрены такие API как select, pselect, poll и epoll [3].

Целью данной курсовой работы является разработка классического статического сервера для отдачи контента с диска на основе технологий pre-fork и epoll.

Для достижения поставленной цели необходимо решить следующие задачи:

— проанализировать способы проектирования многопользовательских серверов, изучить предоставляемые Unix-системами API для их создания;
— спроектировать статический сервер на основе архитектуры pre-fork с использованием epoll;
— реализовать сервер и протестировать разработанное программное обеспечение;
— провести сравнение результатов нагрузочного тестирования разработанного сервера с nginx.

1 Аналитический раздел

1.1 Клиент-серверная архитектура

Клиент-серверная архитектура является фундаментальным понятием, используемым для организации взаимодействия процессов (возможно, запущенных на разных вычислительных машинах).

Сервер – процесс, способный принимать запросы, обрабатывать их и, при наличии и необходимости, возвращать ответ. Клиент – процесс, который отправляет запросы. Таким образом, сервер предоставляет «услуги», а клиент их запрашивает, инициируя коммуникацию.

Модель клиент-серверной архитектуры представлена на рисунке 1.

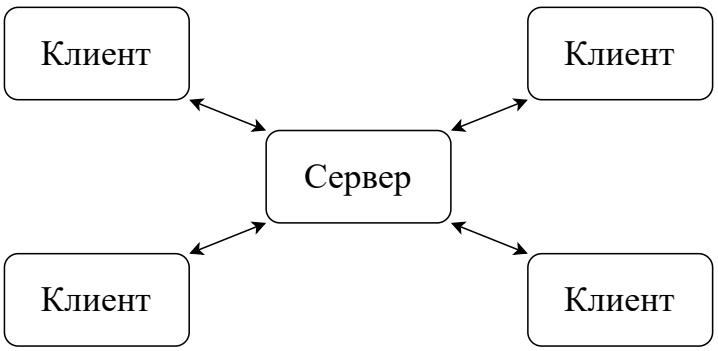

Рисунок 1 – Модель клиент-серверной архитектуры

1.2 Сокеты

Сокет – абстракция конечной точки соединения (взаимодействия). Они делают возможной коммуникацию процессов как на одной, так и на разных вычислительных машинах.

Подобно тому как для работы с файлами приложения используют дескрипторы файлов, для работы с сокетами они используют дескрипторы сокетов. В

UNIX дескрипторы сокетов реализованы так же, как дескрипторы файлов. Разница между ними заключается в множестве допустимых операций, применимых к этим файлам.

Для коммуникации процессов по сети используется понятие порта – конечной точки соединения, обозначаемой 16-битным целым числом. Порты используются для идентификации процесса, которому предназначается пакет. Определение процесса по номеру порта происходит в ядре операционной системы в ходе операции, называемой «демультиплексированием». Это определение успешно, если некоторый процесс «связал» свой сокет с данным портом.

Сокет создается системным вызовом `socket()`. Его прототип представлен в листинге 1.

Листинг 1 – Прототип системного вызова socket()

```
1 #include <sys/socket.h>
2 int socket(int domain, int type, int protocol);
```

Аргумент domain определяет природу взаимодействия, включая формат адреса. Например, домен Интернета IPv4 определяется константой `AF_INET`, а домен UNIX – `AF_UNIX`. Тип сокета (аргумент type) определяет его характеристики взаимодействия. Например, тип `SOCK_DGRAM` не ориентирован на создание логического соединения, допускает отправку сообщений только фиксированной длины и не гарантирует доставку сообщений. Тип `SOCK_STREAM` ориентирован на создание логического соединения, упорядоченность передачи данных и гарантирует их доставку. В качестве протокола (аргумент protocol), как правило, указывается 0, что означает использование протокола по умолчанию. Так, для `SOCK_DGRAM` из домена `AF_INET` это UDP, а для `SOCK_STREAM` из домена `AF_INET` – TCP.

1.2.1 Передача данных

Взаимодействие с использованием сокетов осуществляется по модели клиент-сервер. Последовательность системных вызовов при использовании протокола с установлением соединения (TCP) представлена на рисунке 2.

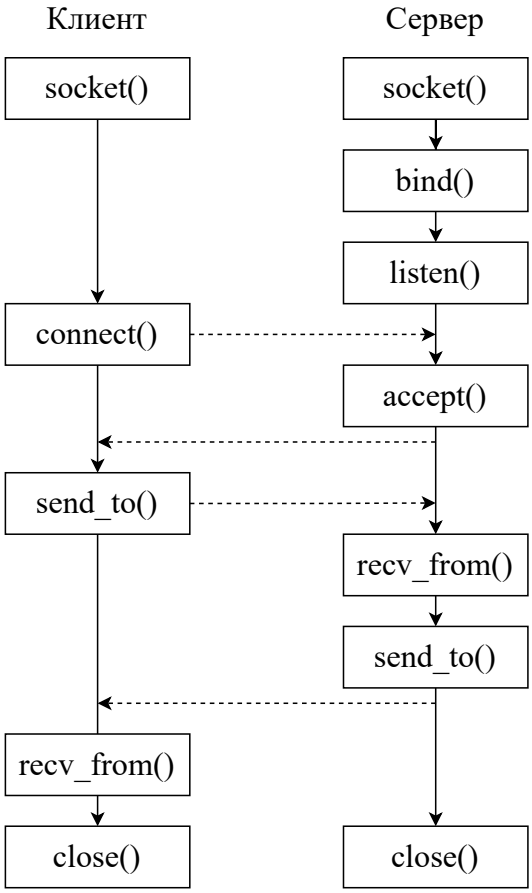

Рисунок 2 – Передача сообщения от клиента серверу

Системный вызов `bind()` используется для назначения сокету локального адреса. Для сетевых сокетов определена структура `struct sockaddr_in`, в которой явно определяется порт и IP-адрес.

Вызов `connect()` устанавливает соединение по адресу, который передается в функцию в качестве аргумента.

Когда сервер получил запрос на соединение, то будет выполнен системный вызов `accept()` – принять этот запрос. Данный системый вызов создает копию исходного сокета и возвращает дескриптор файла нового сокета. В

результате исходный сокет остается в состоянии «listen», а копия – в состоянии «connected». Дублирование сокетов при принятии соединения дает возможность серверу принимать новые соединения без необходимости закрывать ранее принятые.

1.3 Мультиплексирование

Традиционный способ написания серверов – использовать главный поток, заблокированный на системном вызове `accept()` в ожидании новых подключений. Как только приходит новый запрос на соединение, сервер создает новый процесс системным вызовом `fork()`. Дочерний процесс обрабатывает запрос, а родительский (главный) готов снова принимать запросы на подключение.

Чтобы избежать создания нового процесса под каждый запрос, что затратно по ресурсам, используют мультиплексирование запросов средствами API `select`, `poll`, `pselect` и `epoll`.

1.3.1 select

Системный вызов `select()` позволяет программе отслеживать несколько файловых дескрипторов в ожидании, пока один или несколько из них станет «готов» к какого-то вида операции ввода-вывода. Файловый дескриптор считается готовым, если соответствующую операцию ввода-вывода можно произвести без блокировки.

Но поскольку `select()` проектировался до появления концепции неблокирующего ввода-вывода, ряд трудностей делает его использование в современных системах нецелесообразным:

— Для выяснения того, какой именно дескриптор сгенерировал событие, необ-

ходимо вручную опросить их все с помощью `FD_ISSET`, что приводит к излишним затратам ресурсов;

— Максимальное количество одновременно наблюдаемых дескрипторов ограниченно константой `FD_SETSIZE`, которая равна 1024;

— Закрытие дескриптора сокета, отслеживаемого API `select()`, не в главном потоке, приводит к неопределенному поведению;

— Невозможно динамически менять набор наблюдаемых событий;

— Отдельно необходимо вычислять наибольший дескриптор и передавать его отдельным параметром.

1.3.2 pselect

Как и `select()`, `pselect()` ждет изменения статуса нескольких файловых дескрипторов. Эти функции идентичны, за исключением трех отличий между ними:

1) Функция `select` использует время ожидания, которое задано в структуре `struct timeval` (с секундами и микросекундами), тогда как `pselect` использует `struct timespec` (с секундами и наносекундами).
2) Функция `select` может обновить параметр `timeout`, который показывает сколько времени прошло. Функция `pselect` не изменяет этот параметр.
3) Функция `select` не имеет параметра `sigmask`, и т.о. ведет себя также как функция `pselect` вызванная с этим параметром, установленным в `NULL`.

1.3.3 poll

После появления необходимости писать высоконагруженные сервера был спроектирован API `poll()`, учитывающий недостатки `select()`:

— Ограничение в 1024 файловых дескриптора отсутствует;

— Наблюдаемые структуры лучше структурированы;

Однако и `poll()` не лишен недостатков:

— Как и при использовании `select()`, невозможно определить какие именно дескрипторы сгенерировали события без полного прохода по всем наблюдаемым структурам и проверки в них поля revents;

— Как и при использовании `select()`, нет возможности динамически менять наблюдаемый набор событий.

1.3.4 epoll

Данный API появился как логическое продолжение `select()` и `poll()`. От них он отличается тем, что позволяет добавлять, удалять и модифицировать дексрипторы и события в наблюдаемом списке.

Последовательность работы с `epoll` следующая.

1) Создать дескриптор `epoll` с помощью вызова `epoll_create`;
2) Инициализировать структуру `epoll_event` нужными событиями и указателями на контексты соединений;
3) Вызвать `epoll_ctl` посредством макроса `EPOLL_CTL_ADD` для добавления дескриптора в список наблюдаемых;
4) Вызвать `epoll_wait()` для ожидания событий с указанием максимального числа событий, которое можно получить за раз;
5) Обработать полученные события.

Важным отличием является отсутствие необходимости просматривать весь список отслеживаемых дескрипторов.

Достоинства `epoll` следующие:

— Нет необходимости просматривать полный список структур в поисках той, возможно, одной, где сработало ожидаемое событие.

— Есть возможность добавлять или удалять сокеты из списка в любое время. Также можно модифицировать наблюдаемые события.

— Можно завести сразу несколько потоков, ожидающих события из одной и той же очереди с помощью `epoll_wait`.

Несмотря на описанные улучшения, в некоторых ситуациях использование `epoll` нецелесообразно. Недостатки `epoll` следующие:

— Изменение флагов событий происходит средствами лишнего системного вызова `epoll_ctl`, что добавляет лишнее переключение контекста;

— Для каждого нового соединения необходимо вызвать `accept()` и `epoll_ctl` – это два системных вызова. В случае использования poll вызов будет лишь один. При очень коротком времени жизни соединения переключения контекста могут значительно понизить производительность.

— Отсутствие переносимости на платформы, отличные от Linux;

1.4 Модели конкурентных серверов

Самая простая модель обработки запросов к серверу – итеративная. В данной модели сервер может может обрабатывать только одного клиента в единицу времени. Остальные клиенты блокируются до тех пор, пока не подойдет их очередь и им не будет возвращен ответ.

Для обработки нескольких запросов одновременно можно использовать следующий подход: при получении запроса на подключение для его обработки выделяется новый поток или процесс. При высокой нагрузке сервер, использу-

ющий данную модель, неэффективен, поскольку, при увеличении числа потоков или процессов, в мультизадачных системах кванты времени для обработки запросов начинают выделяться реже.

Чтобы избежать падения производительности сервера, связанного с затратами ресурсов на создания потока или процесса, используют следующие модели: pre-forking и pre-threading.

Pre-forking – создание пула копий процесса-родителя. Запрос обрабатывается любым свободным дочерним процессом. Таким образом исключаются затраты на создание процесса под каждый запрос. При повышении нагрузки процесс-родитель может увеличить размер пула. Данная модель представлена на рисунке 3.

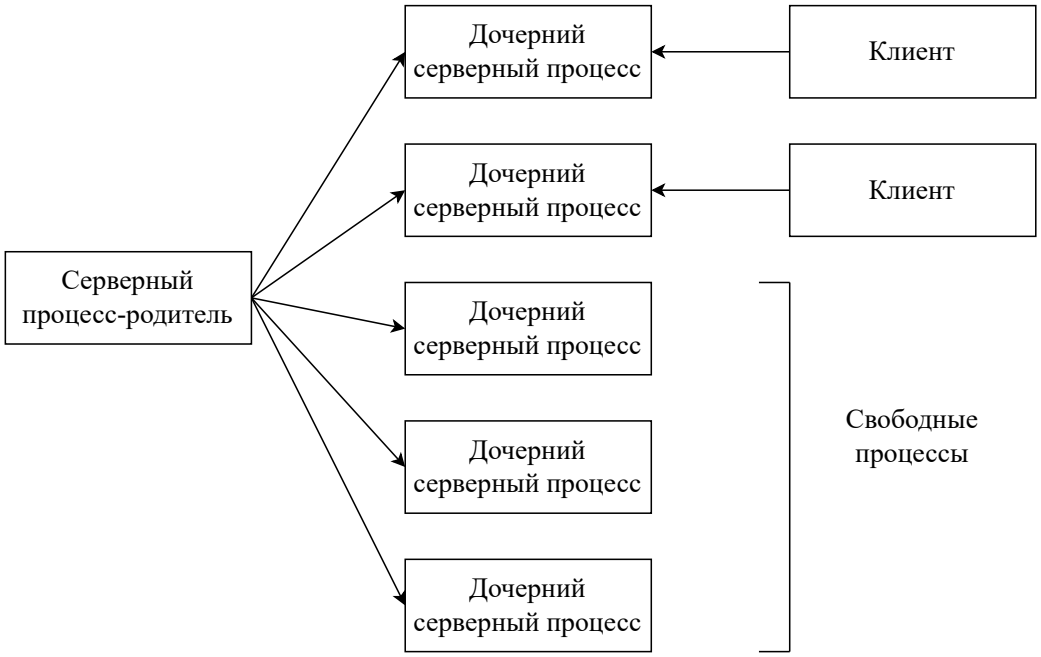

Рисунок 3 – Модель pre-forking

Pre-threading – создание пула потоков-обработчиков запросов. Идея данной модели аналогична pre-forking, только вместо пула процессов используется пул потоков (англ. thread pool). Данная модель представлена на рисунке 4.

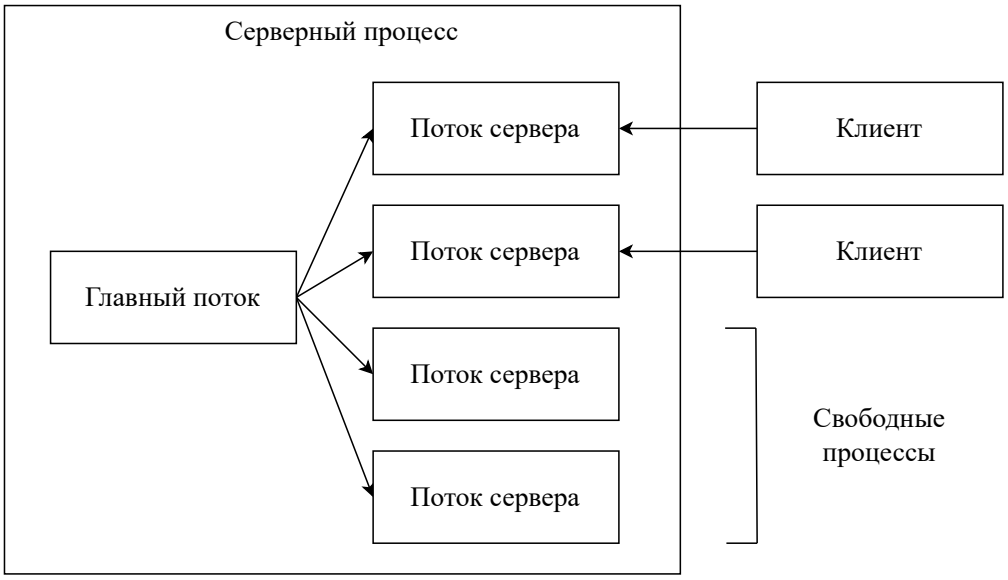

Рисунок 4 – Модель pre-threading

1.5 Проблема «громоподобного стада»

Проблема «громоподобного стада» (англ. thundering herd problem) ставится следующим образом. Имеется M дочерних процессов (или потоков). При старте сервера все из них вызывают `accept()` и блокируются. При первом запросе на подключение все M процессов (потоков) просыпаются. Только один из них перейдет к выполнению задания на обработки, а остальные будут вынуждены снова заблокироваться.

В случае `select` с этой проблемой ничего не поделать. В случае `epoll` данную проблему попытались решить флагом `EPOLLEXCLUSIVE`, который устанавливается вызовом `epoll_ctl`. Данный флаг устанавливает монопольный режим пробуждения для файлового дескриптора `epoll`, ассоциируемого с файловым дескриптором fd. Когда данный флаг установлен, один или несколько файловых дескрипторов `epoll` получат уведомление о событии на `epoll_wait`. Поведение по умолчанию (без флага) – оповещение о событии вообще всех файловых дескрипторов.

Вывод

В данном разделе были проанализированы способы проектирования многопользовательских серверов. Были рассмотрены 4 модели, две из которых признаны наиболее эффективными: pre-threading и pre-forking.

Также были рассмотрены основы сокетов и сетевого стека Linux, а также инструменты мультиплексирования: select, pselect, poll, epoll. Были приведены достоинства и недостатки каждого API, а также рассмотрена проблема «громоподобного стада» и метод её решения в случае epoll.

2 Конструкторский раздел

2.1 Схемы алгоритмов

На рисунке 5 представлен основной алгоритм работы родительского процесса сервера.

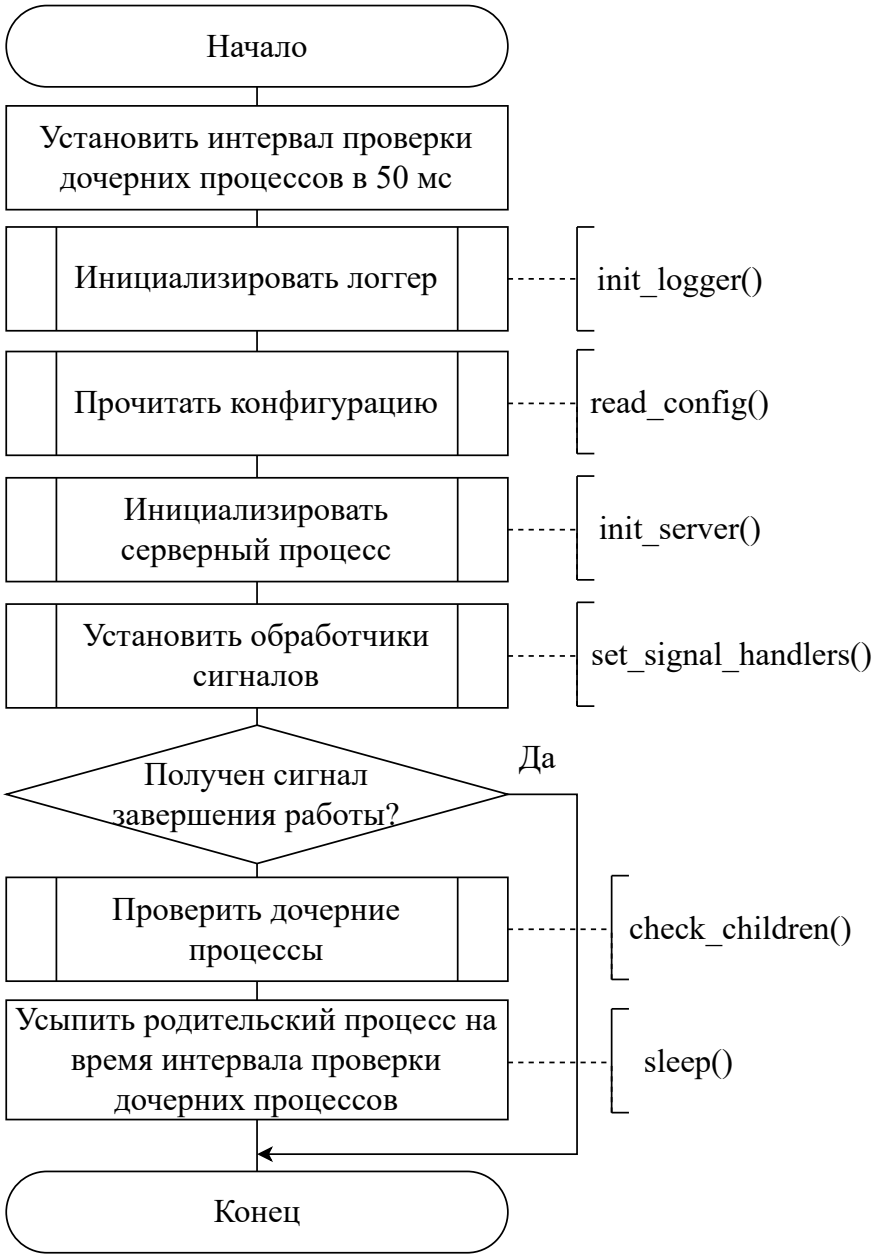

Рисунок 5 – Алгоритм работы процесса-родителя

На рисунке 6 представлен алгоритм функции инициализации сервера.

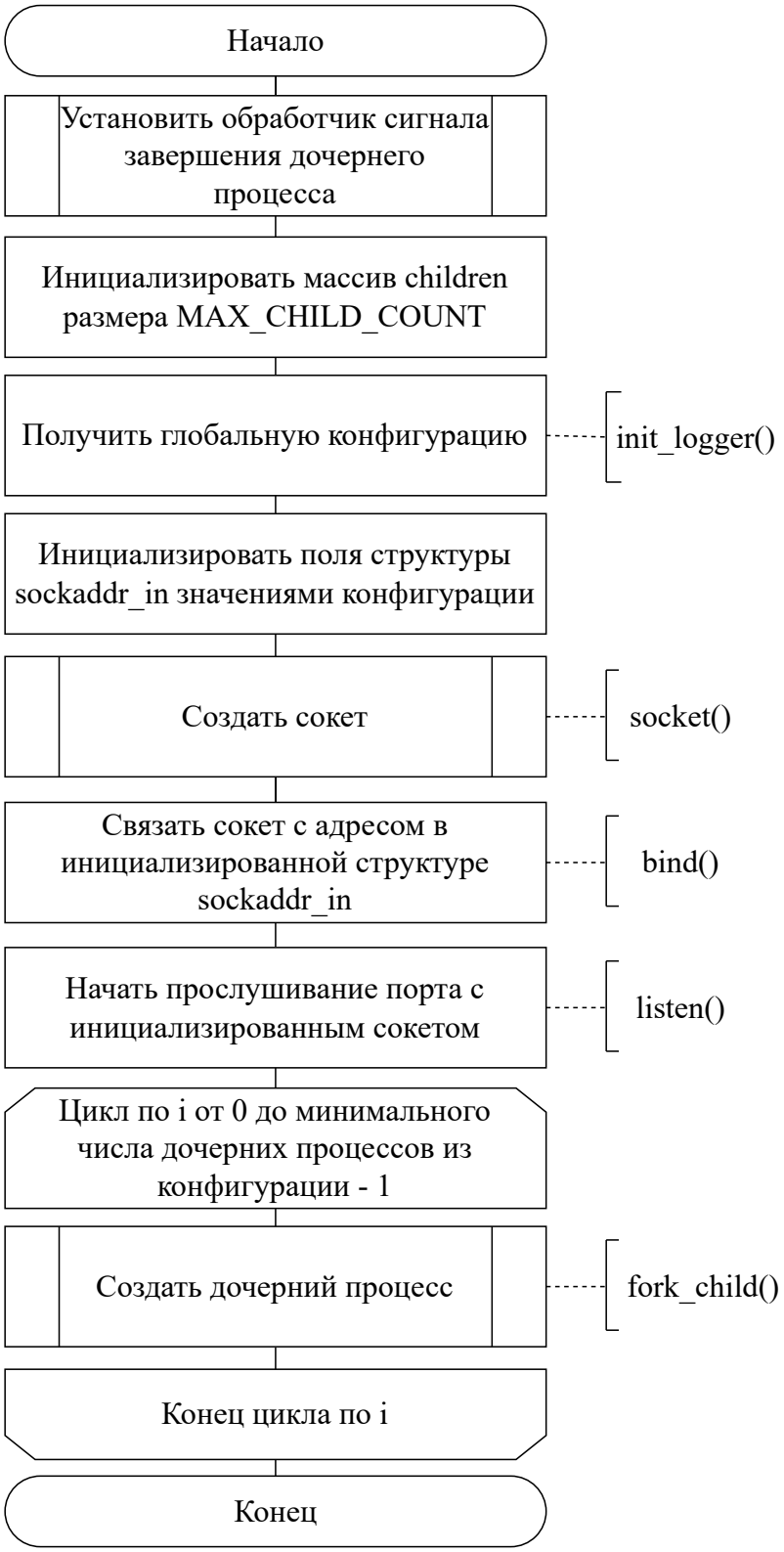

Рисунок 6 – Алгоритм функции init_server()

На рисунке 7 представлен алгоритм создания дочернего процесса сервера.

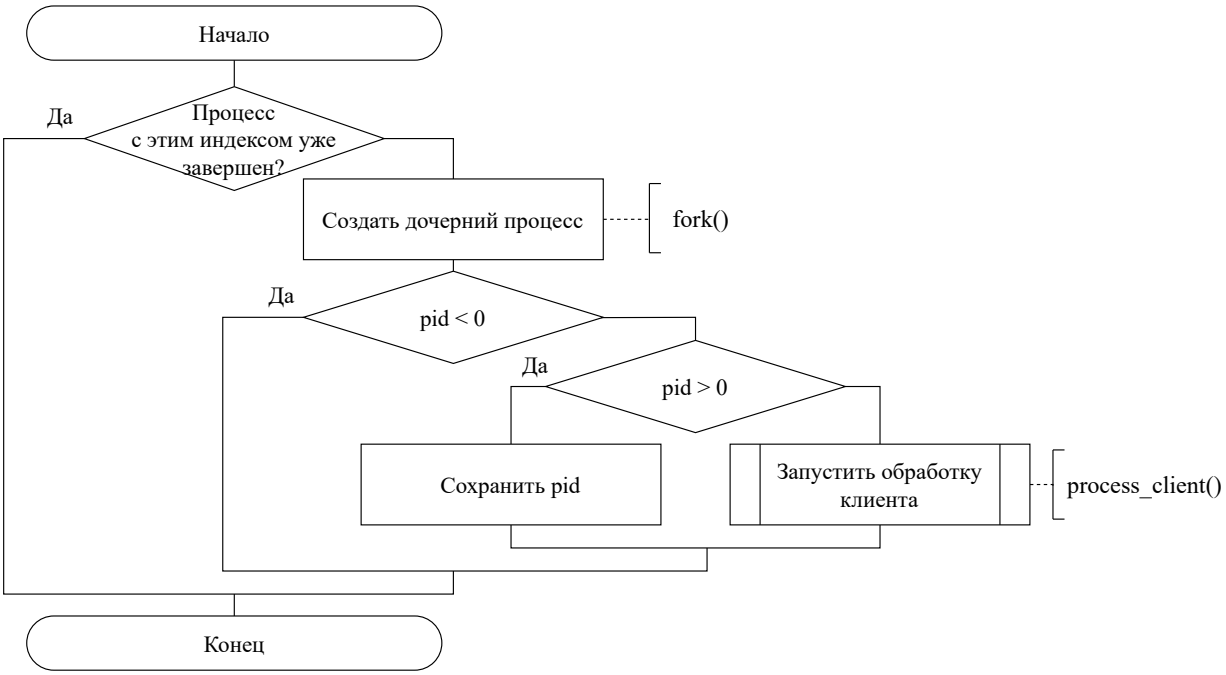

Рисунок 7 – Алгоритм функции fork_child()

На рисунке 8 представлен алгоритм функции инициализации дочернего процесса.

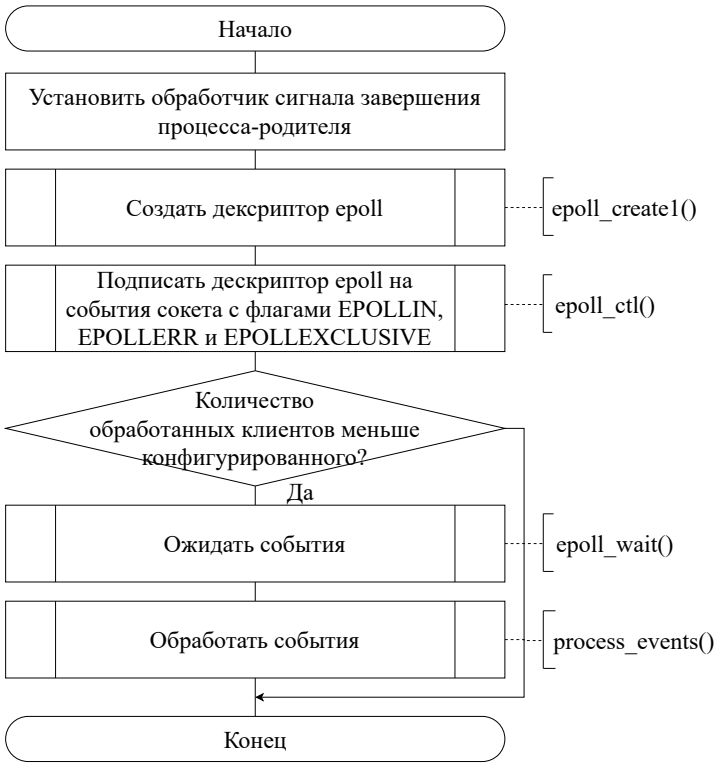

Рисунок 8 – Алгоритм функции process_client()

На рисунке 9 представлен алгоритм функции обработки события в виде запроса.

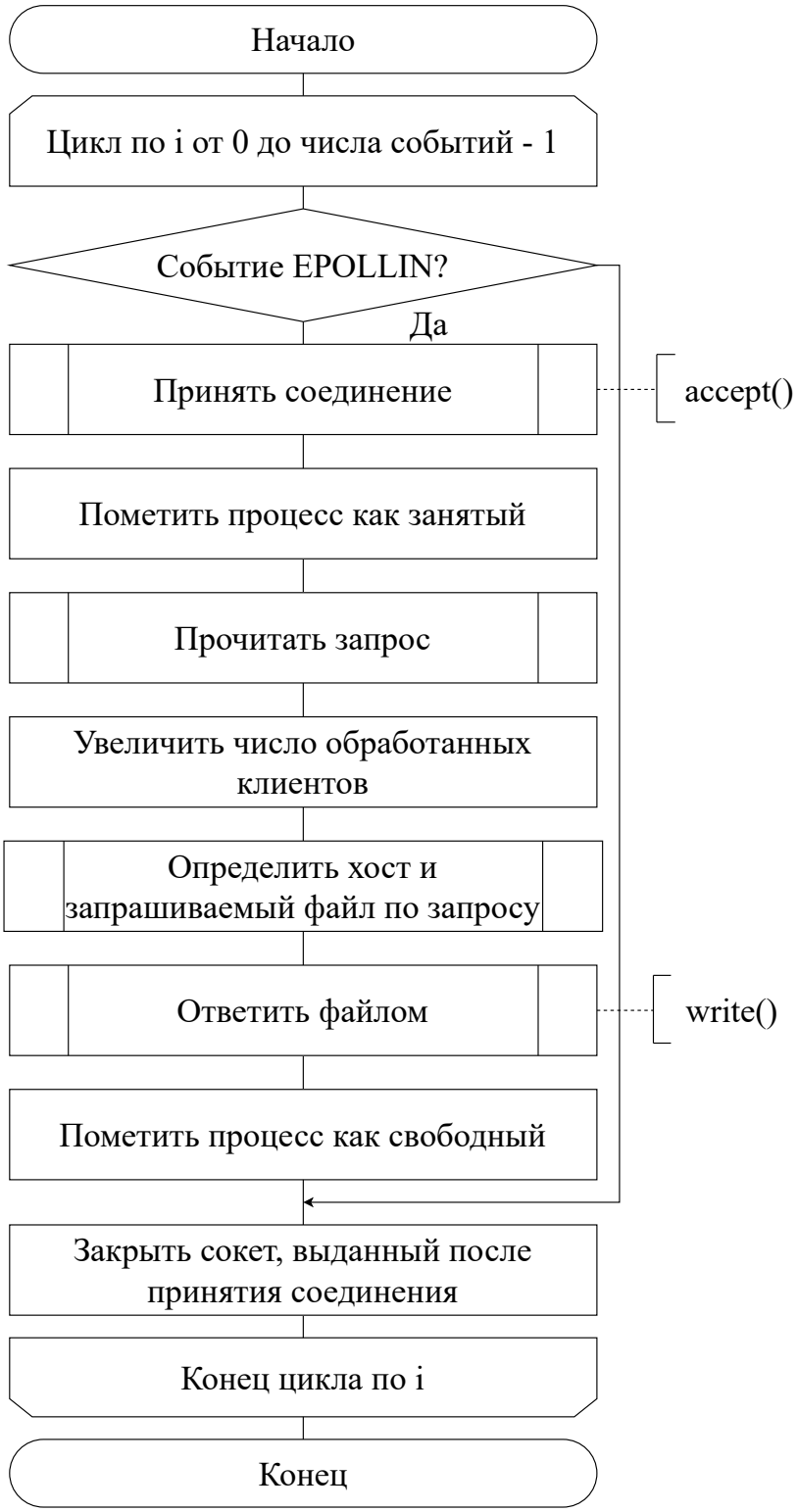

Рисунок 9 – Алгоритм функции process_events()

На рисунке 10 представлен алгоритм функции регулярной проверки пула процессов.

Рисунок 10 – Алгоритм функции check_children()

3 Технологический раздел

3.1 Средства реализации системы

3.1.1 Выбор СУБД

В качестве наиболее подходящей системы управления базами данных выбрана система PostgreSQL [30] – объектно-реляционная СУБД с открытым кодом. Выбор основан на исследованиях [31], которые утверждают, что самыми быстрыми по операции чтения реляционными СУБД являются MySQL [32], PostgreSQL и SQL Server [33]. При этом SQL Server, в отличие от MySQL и PostgreSQL является коммерческой СУБД, что является препятствием для её использования в данном проекте. Что касается MySQL, то в данной СУБД отсутствует возможность создания табличных функций [34], что не позволит реализовать спроектированные алгоритмы.

Что касается кеширующей базы данных, выбор пал на In-Memory Database (IMDB) Redis [40]. Согласно тем же исследованиям [31], она является одной и самых быстрых NoSQL СУБД, и, поскольку не предъявляются требования к наличию хранимых процедур для кеширующей базы данных, Redis предоставляет весь необходимый функционал.

3.1.2 Выбор языка программирования

Во многих современных языках программирования реализованы библиотеки для работы с PostgreSQL. Не исключением является и язык C# [36], выбранный в качестве основного языка программирования в данном проекте. Данный выбор связан с тем, что C# – один из наиболее активно развивающихся [35]

кросс-платформенных языков программирования с открытым исходным кодом. Множество фреймворков и библиотек на платформе .NET позволяют решить практически любую задачу.

Однако для решения поставленной задачи требуются средства обработки естественного языка. Язык C# не обладает достаточным набором средств NLP. Таким образом, часть функций необходимо реализовать на другом языке программирования. Для данных целей наиболее удачно подходит язык Python [37], библиотеки для которого предоставляют огромные возможности для анализа данных – в том числе, и лингвистических.

3.1.3 Выбор средств разработки

В качестве основной среды разработки выбрана среда JetBrains Rider 2022.3.2 [38], поскольку она предоставляет обширный набор возможностей по написанию, тестированию и отладке приложений на языке C#, а также встроенные статические и динамические анализаторы и отдельное окно для подключения к базам данных. Дополнительным преимуществом является то, что данная среда является бесплатной для студентов.

Для разработки компонента обработки лингвистических данных была выбрана среда JetBrains PyCharm 2023.1 [39]. Как и Rider, PyCharm является бесплатной для студентов и предоставляет полный набор инструментов для разработки приложений на языке Python.

3.2 Реализация базы данных

Для инициализации базы данных написаны SQL-сценарии, выполняющиеся при старте СУБД. Сценарий создания таблиц представлен в листинге **??**.

Ограничения на значения столбцов таблиц представлены в листинге **??**.

Разработанная хранимая функция basic_search представлена в листинге **??**. Остальные разработанные функции можно найти в приложении А.

Таблицы country и language изначально заполнятся данными из файлов Countries.csv и Languages.csv. Сценарий заполнения этих таблиц представлен в листинге **??**.

Реализованная ролевая модель представлена в листинге **??**.

3.3 Реализация интерфейса для взаимодействия с базой данных

Интерфейс доступа к базе данных реализован в виде Web API – набора конечных точек соединения на базе фреймворка ASP.NET Core [41]. Для удобства взаимодействия с сервером написана Swagger [42] документация на все контроллеры и модели системы. На основе написанной документации доступен Swagger UI, позволяющий интерактивно взаимодействовать с API. Начальная страница Swagger UI разработанного приложения представлена на рисунке 11.

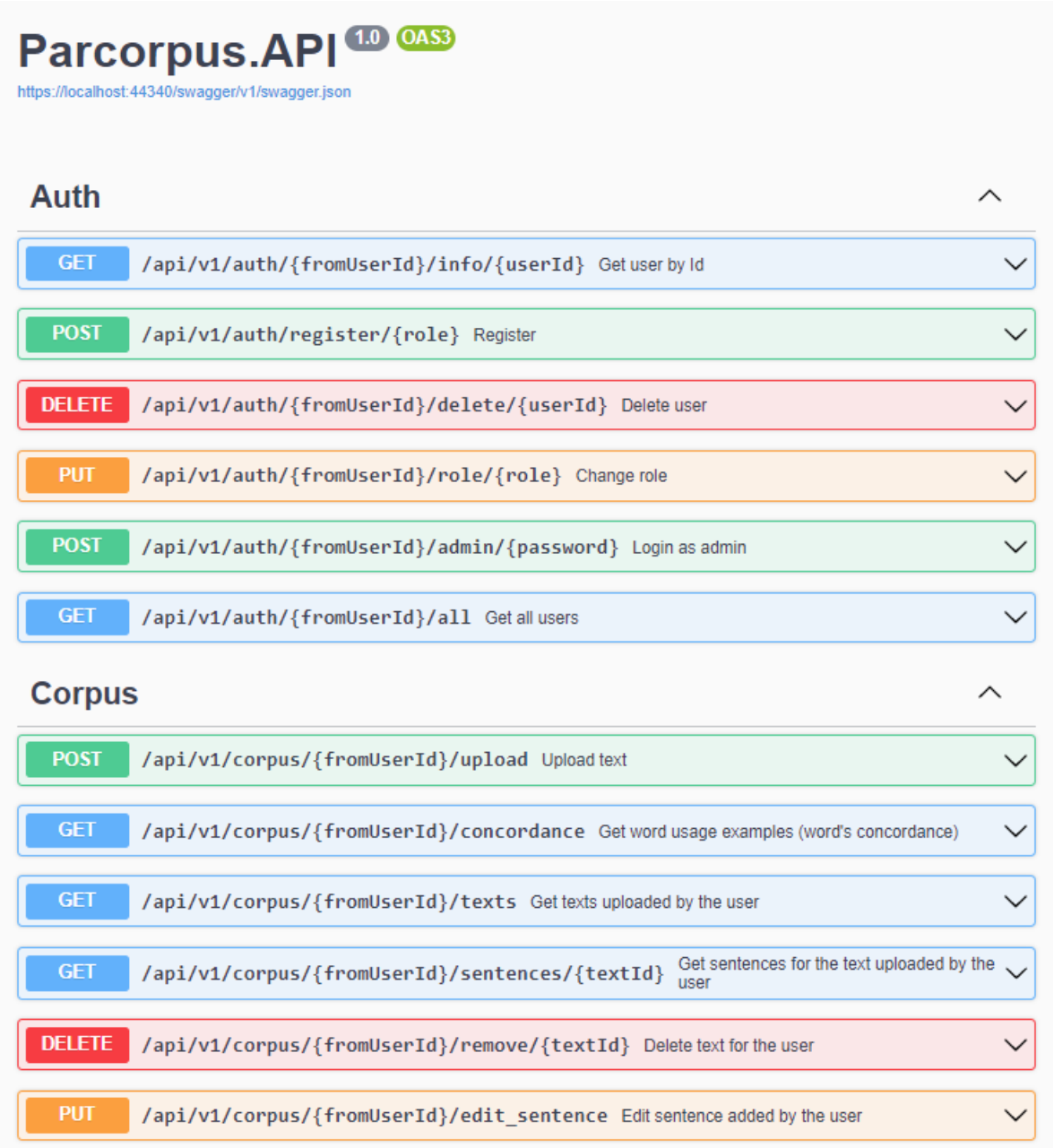

Рисунок 11 – Swagger UI

Чтобы сделать какое-либо действие, необходимо выбрать секцию с необходимой конечной точкой и ввести необходимые данные. Пример регистрации пользователя представлен на рисунке 12.

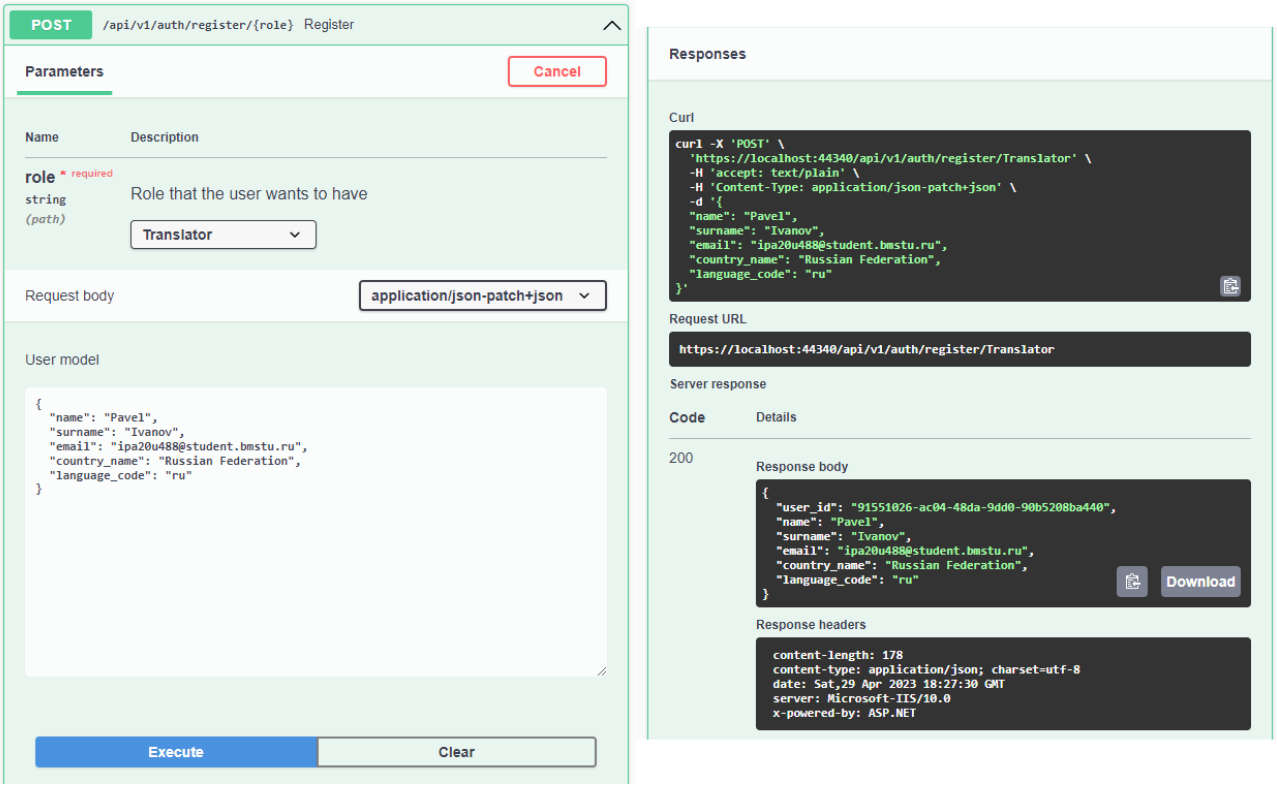

Рисунок 12 – Пример регистрации пользователя

Пример получения примеров употребления слова «и» на английском языке представлен на рисунке 13. При данном запросе прочие фильтры отсутствуют, поэтому некоторым поля результата присвоено значение null.

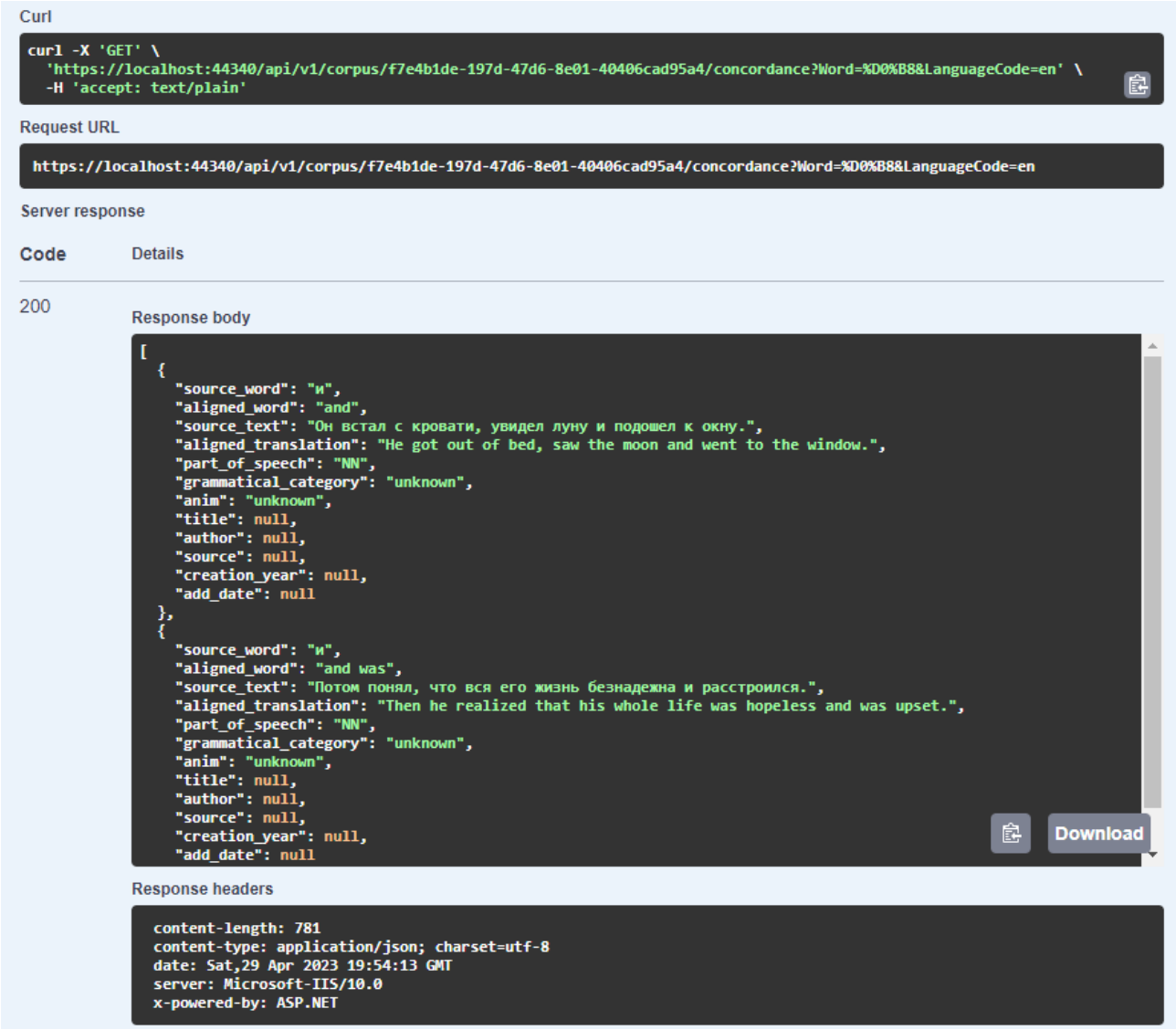

Рисунок 13 – Пример получения примеров употребления слова

3.4 Логирование действий пользователей

Для логирования действий пользователей используется платформа NLog – бесплатный фреймворк для журналирования, совместимый со стеком ELK (Elasticsearch, Logstash, Kibana) [43]. Таким образом, генерируемые логи можно использовать в поисковых и аналитических движках. Фрагмент файла конфигурации представлен в листинге 2.

Листинг 2 – Фрагмент файла NLog.config

```
<target name="asyncAllfile"
        xsi:type="AsyncWrapper"
        timeToSleepBetweenBatches="0"
        overflowAction="Discard"
        batchSize="300" >
    <!-- write logs to file  -->
    <target name="allfile"
            xsi:type="File"
            encoding="utf-8"
            fileName="\${basedir}/log/\${shortdate}.log"
            layout="\${longdate}|\${var:logId}\${guid}|
                    \${level:uppercase=true}|\${var:machineName}|
                    \${message}|
                    \${exception:maxInnerExceptionLevel=7}|
                    \${callsite}"
            archiveFileName="\${basedir}/log/archives/{#}.log"
            archiveEvery="Day"
            archiveNumbering="Date"
            archiveDateFormat="yyyy-MM-dd"
            concurrentWrites="true"/>
</target>
```

Логи сохраняются в txt файлах в папке log с именем, соответствующем текущей дате в формате «yyy-MM-dd». Автоматическое архивирование логов происходит каждый день. Формат записи сообщения в журнале предусматривает запись о дате, уникальном идентификаторе лога, уровне логирования, названии сервера, исключении (с максимальным уровнем вложенности 7) и компоненте, в котором происходит журналирование.

Пример сообщений в журнале при регистрации пользователя представлен в листинге 3.

Листинг 3 – Фрагмент логов при регистрации пользователя

```
2023-04-30 11:37:40.7594|LogId:cf0cc6a99e144680a79cf6727e9c7054|
INFO|Server|Request starting HTTP/2 POST
https://localhost:44340/api/v1/auth/register/Translator
   application/json-patch+json 150||
Microsoft.AspNetCore.Hosting.HostingApplicationDiagnostics.LogRe-
   questStarting
...
2023-04-30 11:37:44.5283|LogId:0417512af22c409586a443f95579b39e|
INFO|Server|Executed DbCommand (72ms)
   [Parameters=[@__countryName_0='?'], CommandType='Text',
   CommandTimeout='30']
SELECT c.country_id, c.name
FROM parcorpusdb.country AS c
WHERE c.name = @__countryName_0
LIMIT 1||Microsoft.EntityFrameworkCore.Diagnostics
.EventDefinition`6.Log

2023-04-30 11:37:44.6832|LogId:6c5d01f84b954671ad9ac7e057cbed29|
INFO|Server|Executed DbCommand (8ms)
   [Parameters=[@__languageShortName_0='?'], CommandType='Text',
   CommandTimeout='30']
SELECT l.language_id, l.full_name, l.short_name
FROM parcorpusdb.language AS l
WHERE l.short_name = @__languageShortName_0
LIMIT 1||
Microsoft.EntityFrameworkCore.Diagnostics
.EventDefinition`6.Log

2023-04-30 11:37:45.1292|LogId:37342430a3fa4fdc9bc2fc05177f4ed9|
INFO|Server|Executed DbCommand (55ms) [Parameters=[@p0='?'
   (DbType = Guid), @p1='?' (DbType = Int32), @p2='?', @p3='?',
   @p4='?' (DbType = Int32), @p5='?'], CommandType='Text',
   CommandTimeout='30']
```

Окончание листинга 3

```
INSERT INTO parcorpusdb."user" (user_id, country, email, name,
    native_language, surname)
VALUES (@p0, @p1, @p2, @p3, @p4, @p5);||
Microsoft.EntityFrameworkCore.Diagnostics.EventDefinition`6.Log

2023-04-30 11:37:53.2174|LogId:3ce6e47b6a504f49bc8e497ec696801c|
INFO|Server|Executed DbCommand (17ms) [Parameters=[],
    CommandType='Text', CommandTimeout='30']
create user "5182a73d-194a-4665-9651-a8c862571daf";||
Microsoft.EntityFrameworkCore.Diagnostics.EventDefinition`6.Log

2023-04-30 11:37:53.2891|LogId:a0aaf2140d264e06b72d430597aee1b2|
INFO|Server|Executed DbCommand (64ms) [Parameters=[],
    CommandType='Text', CommandTimeout='30']
grant translator to "5182a73d-194a-4665-9651-a8c862571daf";||
Microsoft.EntityFrameworkCore.Diagnostics.EventDefinition`6.Log
...
2023-04-30 11:37:53.4192|LogId:221418d9b0ba4538b118b2fd5d90a091|
INFO|Server|Request finished HTTP/2 POST
    https://localhost:44340/api/v1/auth/register/Translator
    application/json-patch+json 150 - 200 178
    application/json;+charset=utf-8 12663.1748ms||
Microsoft.AspNetCore.Hosting.HostingApplicationDiagnostics
.LogRequestFinished
```

3.5 Инструкция для развертывания системы

Для развертывания разработанной системы необходимо установить Docker – программное обеспечение для автоматизации развертывания и управления приложениями в средах с поддержкой контейнеризации [44]. Docker доступен как на Windows, так и на Linux или MacOS.

Для запуска системы на компьютере или на сервере необходимо проделать следующие действия:

1) открыть терминал в директории приложения (в ней находятся папки doc, src и test);
2) ввести команду «cd ./src/Scripts && docker-compose up -d».

После нажатия Enter система Docker самостоятельно скачает и установит все необходимые пакеты, а также запустит СУБД и сервисы приложения. Пример вывода консоли при запуске системы представлен в листинге 4.

Листинг 4 – Запуск системы

```
C:\Users\Pavel\Desktop\parcorpus\src\Scripts>
docker-compose up -d
[+] Running 5/5
Network scripts_backend-network Created 0.1s
Container redis                 Started 3.0s
Container aligner               Started 3.2s
Container postgres              Started 3.4s
Container parcorpus-backend     Started
```

После запуска системы её записи журнала доступны на локальной машине и находятся в папке /src/Parcorpus/Parcorpus.API/logs. Изменение настроек (например, строки подключения к базе данных или кеширующей СУБД) доступно в файле /src/Parcorpus/Parcorpus.API/appsettings.json. При изменении данного файла будет также изменен файл в контейнере.

После запуска системы взаимодействие может с ней осуществляется по адресу http://localhost:802/. Список доступных конечных точек соединения по данному адресу представлен в таблице 1.

Таблица 1 – Конечные точки соединения

Конечная точка	Параметры	Описание
GET: /api/v1/auth/ {fromUserId}/info/ {userId}	fromUserId – идентификатор пользователя, который совершает запрос; userId – идентификатор пользователя, о котором запрашивается информация.	Получение информации о пользователе
DELETE: /api/v1/auth/ {fromUserId}/delete/ {userId}	fromUserId – идентификатор пользователя, который совершает запрос; userId – идентификатор удаляемого пользователя.	Удаление пользователя
PUT: /api/v1/auth/ {fromUserId}/role/{role}	fromUserId – идентификатор пользователя, который совершает запрос; role – новая роль пользователя.	Смена роли
POST: /api/v1/auth/register/{role}	role – роль регистрируемого пользователя; Тело запроса (application/json-patch+json): UserRegistrationDto.	Регистрация нового пользователя

Продолжение таблицы 1

Конечная точка	Параметры	Описание
POST: /api/v1/auth/ {fromUserId}/admin/ {password}	fromUserId – идентификатор пользователя, который совершает запрос; password – пароль.	Авторизация администратора
GET: /api/v1/auth/ {fromUserId}/all	fromUserId – идентификатор пользователя, который совершает запрос.	Получение информации о всех пользователях
POST: /api/v1/corpus/ {fromUserId}/upload	fromUserId – идентификатор пользователя, который совершает запрос. Тело запроса (multipart/form-data): InputTextDto.	Загрузка нового текста
GET: /api/v1/corpus/ {fromUserId}/ concordance	fromUserId – идентификатор пользователя, который совершает запрос. Тело запроса (multipart/form-data): ConcordanceQueryDto.	Поиск примеров употребления слова
GET: /api/v1/corpus/ {fromUserId}/texts	fromUserId – идентификатор пользователя, который совершает запрос;	Получение всех текстов пользователя

Окончание таблицы 1

Конечная точка	Параметры	Описание
GET: /api/v1/corpus/ {fromUserId}/sentences/ {textId}	fromUserId – идентификатор пользователя, который совершает запрос; textId – идентификатор текста.	Получение всех предложений добавленного пользователем текста
DELETE: /api/v1/corpus/ {fromUserId}/remove/ {textId}	fromUserId – идентификатор пользователя, который совершает запрос; textId – идентификатор текста.	Удаление текста
PUT: /api/v1/corpus/ {fromUserId}/ edit_sentence	fromUserId – идентификатор пользователя, который совершает запрос; Тело запроса (application/json-patch+json): SentenceDto.	Редактирование предложения

Упомянутые объекты передачи данных (англ. «Data transfer object», «DTO») имеют структуру, представленную в листингах 5–9.

Листинг 5 – UserRegistrationDto

```
1 public class UserRegistrationDto
2 {
3     /// <summary>  Имя пользователя  </summary>
4     /// <example>Pavel</example>
5     [JsonProperty("name")]
6     public string Name { get; set; }
7
8     /// <summary> Фамилия пользователя </summary>
```

Окончание листинга 5

```
    /// <example>Ivanov</example>
    [JsonProperty("surname")]
    public string Surname { get; set; }

    /// <summary> Электронная почта пользователя </summary>
    /// <example>ipa20u488@student.bmstu.ru</example>
    [JsonProperty("email")]
    public string Email { get; set; }

    /// <summary> Название страны пользователя </summary>
    /// <example>Russian Federation</example>
    [JsonProperty("country_name")]
    public string CountryName { get; set; }

    /// <summary>
    /// Сокращенное название родного языка пользователя
    /// </summary>
    /// <example>ru</example>
    [JsonProperty("language_code")]
    public string LanguageCode { get; set; }
}
```

Листинг 6 – InputTextDto

```
public class InputTextDto
{
    /// <summary> Сокращенное название исходного языка </summary>
    /// <example>ru</example>
    public string SourceLanguageCode { get; set; }

    /// <summary> Сокращенное название целевого языка </summary>
    /// <example>en</example>
    public string TargetLanguageCode { get; set; }
```

Окончание листинга 6

```
10      /// <summary> Название текста </summary>
11      public string? Title { get; set; }
12
13      /// <summary> Автор текста </summary>
14      public string? Author { get; set; }
15
16      /// <summary> Источник текста </summary>
17      public string? Source { get; set; }
18
19      /// <summary> Год написания текста </summary>
20      public int? CreationYear { get; set; }
21
22      /// <summary> Перечисление жанров </summary>
23      public IEnumerable<string> Genres { get; set; }
24
25      /// <summary> Файл с исходным текстом </summary>
26      [FromForm(Name="SourceText")]
27      public IFormFile SourceText { get; set; }
28
29      /// <summary> Файл с переведенным текстом </summary>
30      [FromForm(Name="TargetText")]
31      public IFormFile TargetText { get; set; }
32  }
```

Листинг 7 – ConcordanceQueryDto

```
1  public class ConcordanceQueryDto
2  {
3      /// <summary> Слово для поиска </summary>
4      [JsonProperty("word")]
5      public string Word { get; set; }
6
7      /// <summary> Краткое название целевого языка </summary>
```

Окончание листинга 7

```
    /// <example>"ru"</example>
    [JsonProperty("language_code")]
    public string LanguageCode { get; set; }

    /// <summary> Фильтры поиска </summary>
    [JsonProperty("filter")]
    public FilterDto? Filter { get; set; }
}
```

Листинг 8 – FilterQueryDto

```
public class FilterDto
{
    /// <summary> Жанр </summary>
    [JsonProperty("genre")]
    public string? Genre { get; set; }

    /// <summary> Дата начала </summary>
    [JsonProperty("start_date_time")]
    public DateTime? StartDateTime { get; set; }

    /// <summary> Дата окончания </summary>
    [JsonProperty("end_date_time")]
    public DateTime? EndDateTime { get; set; }

    /// <summary> Часть речи </summary>
    [JsonProperty("part_of_speech")]
    public string? Pos { get; set; }

    /// <summary> Грамматическая категория </summary>
    [JsonProperty("gram")]
    public string? Gram { get; set; }

    /// <summary> Одушевленность </summary>
```

Окончание листинга 8

```
24      [JsonProperty("anim")]
25      public string? Anim { get; set; }
26
27      /// <summary> Автор </summary>
28      [JsonProperty("author")]
29      public string? Author { get; set; }
30  }
```

Листинг 9 – SentenceDto

```
1  public class SentenceDto
2  {
3      /// <summary> Id предложения </summary>
4      [JsonProperty("sentence_id")]
5      public int? SentenceId { get; set; }
6
7      /// <summary>
8      /// Новый текст предложения на исходном языке
9      /// </summary>
10     /// <example>I saw an apple</example>
11     [JsonProperty("source_text")]
12     public string? SourceText { get; set; }
13
14     /// <summary> Новый перевод </summary>
15     /// <example>Я увидел яблоко</example>
16     [JsonProperty("aligned_translation")]
17     public string? AlignedTranslation { get; set; }
18 }
```

Общая последовательность действий при работе с системой следующая:

1) Регистрация и получение UserId.

```
curl -X 'POST' \
    'http://localhost:802/api/v1/auth/register/Translator' \
    -H 'accept: text/plain' \
    -H 'Content-Type: application/json-patch+json' \
    -d '{ \
        "name": "Peter", \
        "surname": "Parker", \
        "email": "ipa20u488@student.bmstu.ru", \
        "country_name": "Russian Federation", \
        "language_code": "ru" \
    }'
```

В качестве ответа возвращается JSON, содержащий UserId.

```
{
    "user_id": "9c95e47a-9617-4ab3-b80d-b31055a97f19",
    "name": "Peter",
    "surname": "Parker",
    "email": "ipa20u488@student.bmstu.ru",
    "country_name": "Russian Federation",
    "language_code": "ru"
}
```

2) Запомнив UserId, можно выполнять любые запросы. Например, получение примеров употребления.

```
curl -X 'GET' \
    'http://localhost:802/api/v1/corpus/ \
    9c95e47a-9617-4ab3-b80d-b31055a97f19/concordance? \
    Word=%D0%B8&LanguageCode=en&Filter.Genre=Drama' \
    -H 'accept: text/plain'
```

Полученный ответ:


```
[
    {
        "source_word": "и",
        "aligned_word": "and",
        "source_text": "Он встал с кровати, увидел луну и
                        подошел к окну.",
        "aligned_translation": "He got out of bed, saw the
                           moon and went to the window.",
        "part_of_speech": "NN",
        "grammatical_category": "unknown",
        "anim": "unknown",
        "title": "My example 1",
        "author": "Pavel Ivanov",
        "source": "Pavel Ivanov",
        "creation_year": "2023-01-01",
        "add_date": "2023-04-29"
    }, {
        "source_word": "и",
        "aligned_word": "and was",
        "source_text": "Потом понял, что вся его жизнь
                        безнадежна и расстроился.",
        "aligned_translation": "Then he realized that his
                  whole life was hopeless and was upset.",
        "part_of_speech": "NN",
        "grammatical_category": "unknown",
        "anim": "unknown",
        "title": "My example 1",
        "author": "Pavel Ivanov",
        "source": "Pavel Ivanov",
        "creation_year": "2023-01-01",
        "add_date": "2023-04-29"
    }
]
```

3) Возможна аутентификация в качестве администратора. Для этого необходимо ввести пароль: pavelivanovIU7-65B.

```
curl -X 'POST' \
    'http://localhost:802/api/v1/auth/ \
        9c95e47a-9617-4ab3-b80d-b31055a97f19/admin/ \
        pavelivanovIU7-65B' \
    -H 'accept: */*' \
    -d ''
```

В случае успеха возвращается код 200.

3.6 Тестирование

В результате тестирования ролей получены результаты, представленные в таблице 2. Как видно из полученных результатов, разработанная система соответствует спроектированной ролевой модели. В случае отсутствия прав возвращается ошибка 401. В методах sentences, remove и edit_sentence ошибка 401 возникает в случае обращения к тексту, который добавил другой пользователь (кроме случая администратора).

Таблица 2 – Результаты тестирования разделения ролей

Роль	Конечная точка	Ответ
Translator | Admin	/api/v1/corpus/{fromUserId}/upload	201 / 409
Teacher	/api/v1/corpus/{fromUserId}/upload	401
Teacher | Translator | Admin	/api/v1/corpus/{fromUserId}/concordance	200 / 404
Translator | Translator | Admin	/api/v1/corpus/{fromUserId}/texts	200 / 404
Translator | Teacher	/api/v1/corpus/{fromUserId}/sentences/{textId}	200 / 401 / 404
Admin	/api/v1/corpus/{fromUserId}/sentences/{textId}	200 / 404
Translator | Teacher	/api/v1/corpus/{fromUserId}/remove/{textId}	200 / 401 / 404
Admin	/api/v1/corpus/{fromUserId}/remove/{textId}	200 / 404
Translator | Teacher	/api/v1/corpus/{fromUserId}/edit_sentence	200 / 401 / 404
Admin	/api/v1/corpus/{fromUserId}/edit_sentence	200 / 404
Translator | Teacher	/api/v1/auth/{fromUserId}/info/{userId}	401
Admin	/api/v1/auth/{fromUserId}/info/{userId}	200 / 404

Окончание таблицы 2

Роль	Конечная точка	Ответ
Translator | Teacher	/api/v1/auth/{fromUserId}/ delete/{userId}	401
Admin	/api/v1/auth/{fromUserId}/ delete/{userId}	200 / 404
Translator | Teacher | Admin	/api/v1/auth/{fromUserId}/ admin/{password}	200 / 403
Translator | Teacher	/api/v1/auth/{fromUserId}/all	401
Admin	/api/v1/auth/{fromUserId}/all	200

Вывод

В данном разделе было приведено обоснование выбора средств реализации системы. В качестве основной реляционной СУБД была выбрана система с открытым кодом PostgreSQL, поскольку она является одной из самых быстрых (по операции чтения) СУБД. Для системы кеширования была выбрана In-Memory Database Redis, поскольку она является одной из самых быстрых NoSQL СУБД и предоставляет весь необходимый для решения поставленной задачи функционал.

Для написания интерфейса для взаимодействия с параллельным корпусом был выбран язык C#, а для обработки лингвистических данных язык Python. В результате написано REST API на основе фреймворка ASP.NET Core, предоставляющее набор конечных точек для работы с корпусом. Разработанная система реализует спроектированную ролевую модель, систему журналирования действий пользователя и использование кеша для ускорения получения ответов на выборку данных из корпуса. Тестирование разделения ролей показало, что

разработанная система соответствует спроектированной ролевой модели.

4 Исследовательский раздел

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры, следующие.

— Операционная система Windows 10 [46] x86_64.

— Память 8 Гб 2400 МГц DDR4.

— 1.6 ГГц 4-ядерный процессор Intel Core i5 8265U [47].

Замеры проводились на ноутбуке, включенном в сеть электропитания. Во время проведения исследований ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой выполнения замеров.

4.2 Подготовка исследования

Для выявления зависимости времени исполнения запросов от использования кеширующей СУБД база данных была пополнена текстами на французском и немецком языках. База данных заполнена более чем тридцатью предложениями и более чем 500 словами. На момент начала замеров кеширующая СУБД пуста.

Замеры производятся для двух запросов на получение примеров употребления слова в корпусе.

1) Поиск перевода французского слова «nous» (рус. «мы») на немецкий язык: api/v1/corpus/{uuid}/concordance?Word=nous&LanguageCode=de. Далее обозначим этот запрос как «простой», поскольку он не предполагает фильтрацию.
2) Поиск перевода французского слова «nous» на немецкий язык среди тек-

стов «Описательного» жанра, написанные с 1 января 2000 по 14 мая 2023: api/v1/corpus/{uuid пользователя}/concordance?Word=nous&LanguageCode=de&Filter.Genre=Description&Filter.StartDateTime=01.01.2000&Filter.EndDateTime=05.14.2023. Обозначим этот запрос как «сложный», поскольку он потребует больше соединений таблиц.

4.3 Результаты замеров

На рисунке 14 представлено время выполнения «простого» запроса методом GetConcordance класса LanguageRepository, который выполняет проверку на наличие в кеше данных по запросу и непосредственное выполнение запросов к базе данных.

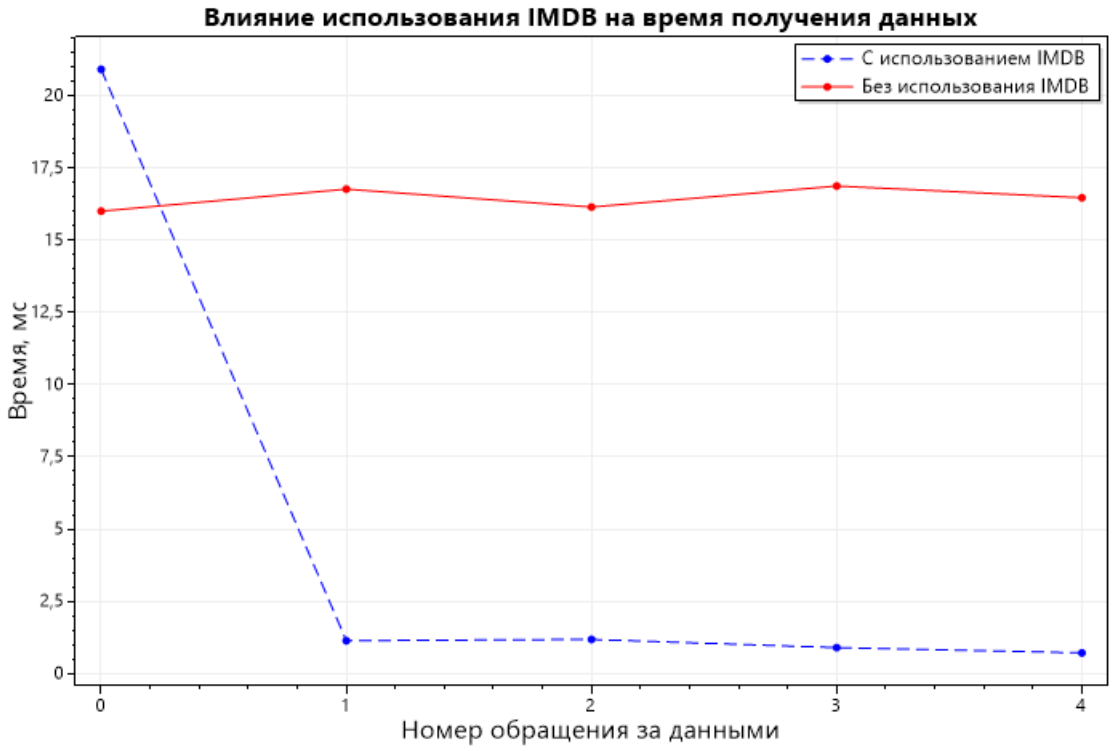

Рисунок 14 – Время выполнения «простого» запроса методом GetConcordance

Как видно из полученных данных, после первого обращения за данными время их получения уменьшается более чем в 20 раз. Время выполнения «простого» запроса без использования кеширующей СУБД не зависит от номера обращения за данными.

Аналогичные результаты получены для «сложного» запроса. Время его выполнения представлено на рисунке 15.

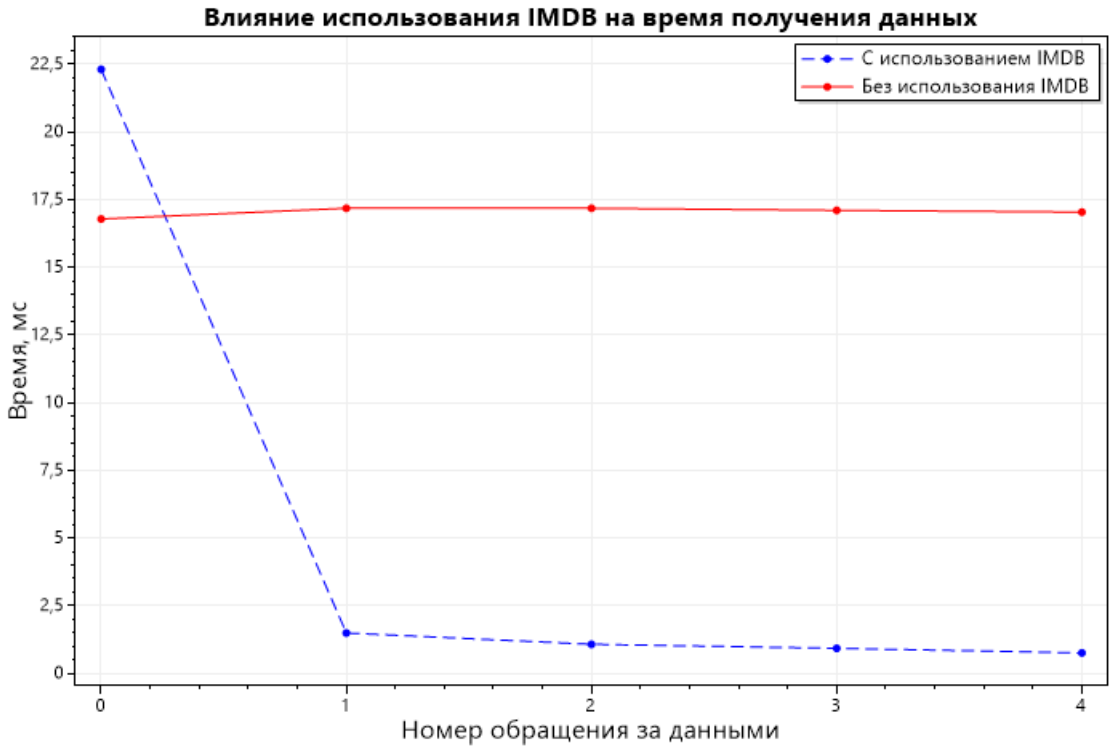

Рисунок 15 – Время выполнения «сложного» запроса методом GetConcordance

Данные, представленные на рисунках 14 и 15, получены вследствие усреднения результатов 500 замеров времени выполнения запросов с одними и теми же входными данными.

С помощью программы Apache JMeter [48] было проведено нагрузочное тестирование системы. Результаты замеров времени ответа системы на «простой» запрос с использованием и без использования кеширующей СУБД представлены на рисунках 16 и 17 соответственно.

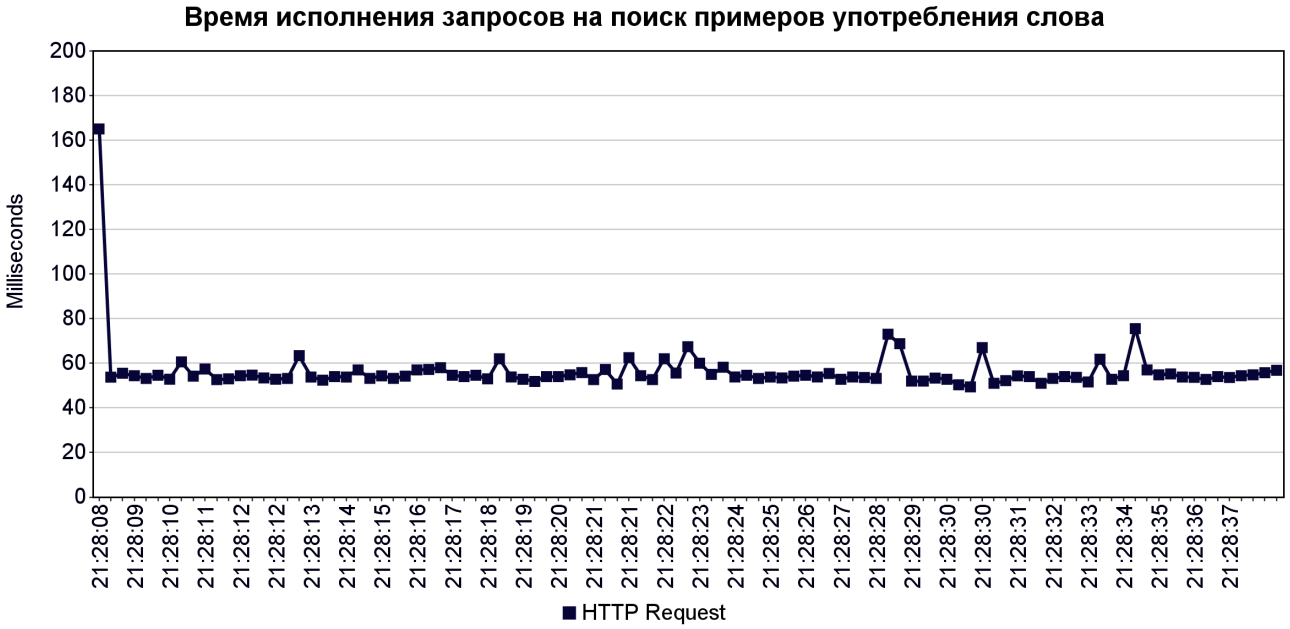

Рисунок 16 – Время выполнения «простого» запроса к системе с использованием кеширующей СУБД

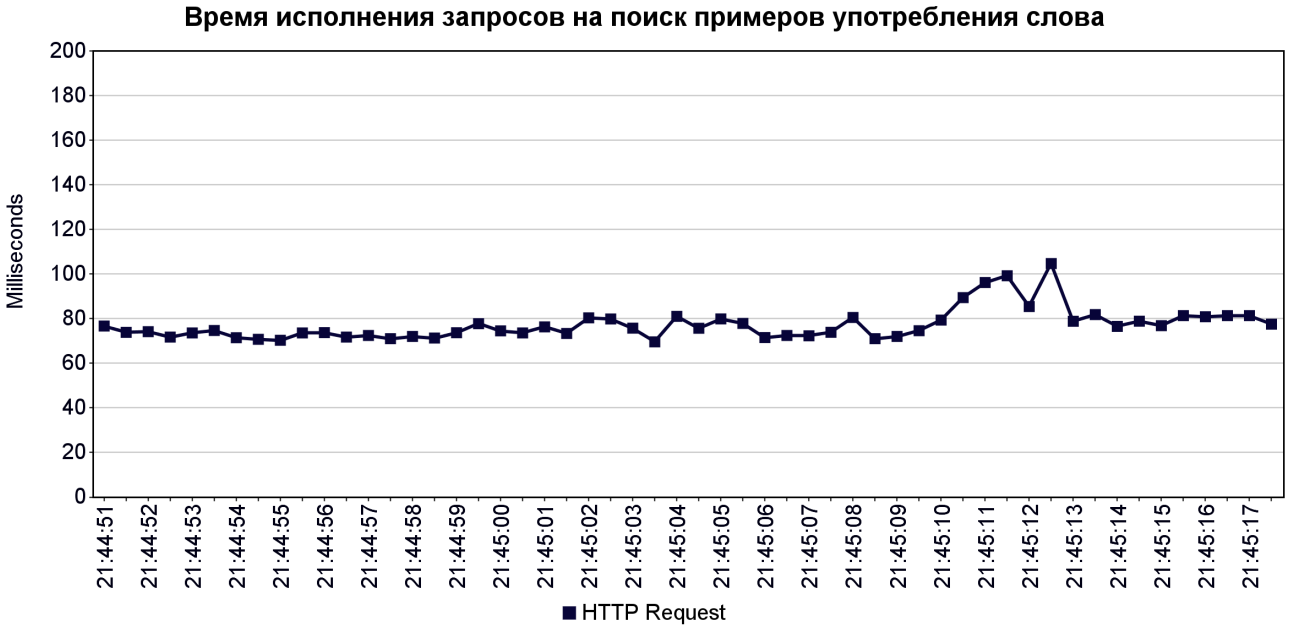

Рисунок 17 – Время выполнения «простого» запроса к системе без использования кеширующей СУБД

На рисунках 18 и 19 представлены аналогичные замеры для «сложного» запроса.

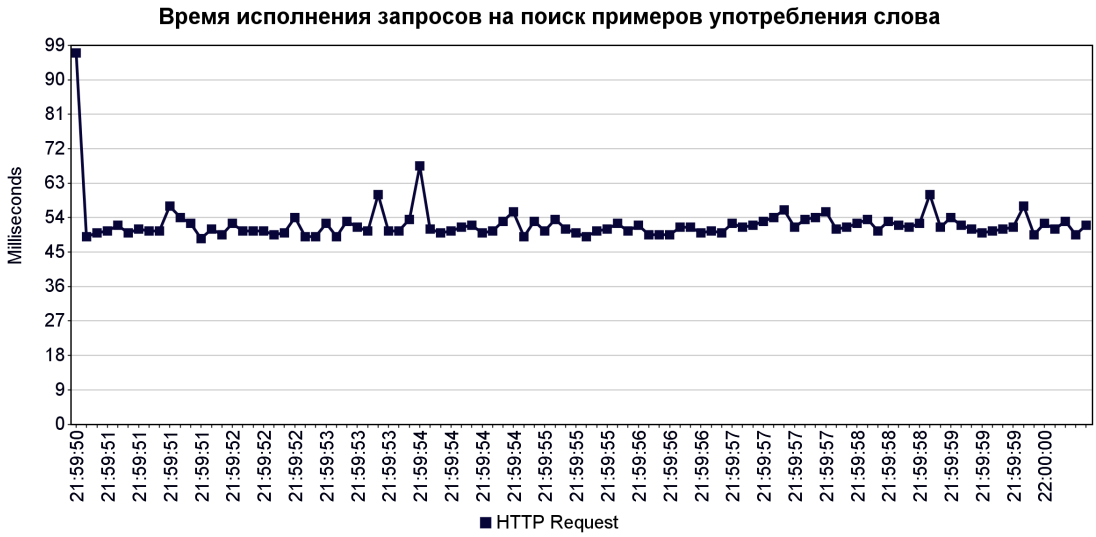

Рисунок 18 – Время выполнения «сложного» запроса к системе с использованием кеширующей СУБД

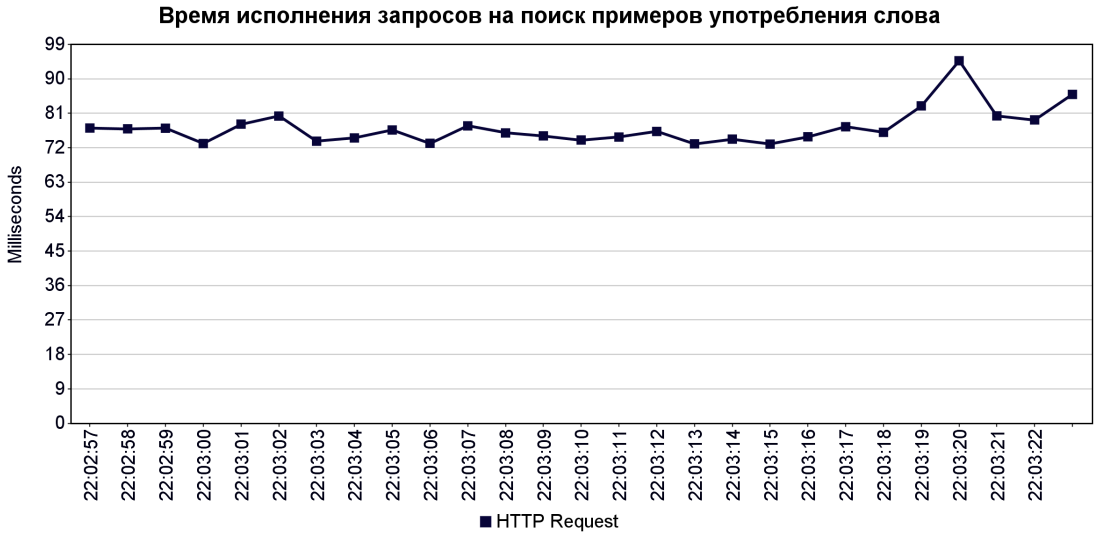

Рисунок 19 – Время выполнения «сложного» запроса к системе без использования кеширующей СУБД

Таким образом, благодаря использованию кеширующей СУБД, время получения ответа как на «простой», так и на «сложный» запрос уменьшается на 25%: с 80 мс до 60 мс. Однако, в случае использования кеширующей СУБД, при первом получении запроса необходимо записать значение в кеш.

Вывод

В результате проведенных замеров и, в том числе нагрузочного тестирования, установлено, что использование кеширующей СУБД позволяет сократить время повторного получения данных на 25%. Однако, в связи с необходимостью записи данных в кеширующую СУБД, время получения ответа на новый запрос увеличивается.

ЗАКЛЮЧЕНИЕ

В данной курсовой работе была проанализирована предметная область параллельных корпусов текстов, были даны определения ключевым терминам и описаны способы применения корпуса. На основе проведенного анализа были сформулированы требования к наполнению корпуса и его возможностям, разработана ролевая модель. В качестве используемой разметки были выбраны морфологическая аннотация и метаразметка. В результате проведенного анализа существующих СУБД было выявлено, что наиболее подходящим для решения поставленной задачи видом СУБД является реляционная база данных, развернутая локально.

Были выделены сущности проектируемой базы данных, представлены ER-диаграмма в нотации Чена и диаграмма проектируемой базы данных. Для всех таблиц были описаны их атрибуты. Разработан алгоритм хранимой функции поиска примеров употребления слова в корпусе, а также алгоритм функции поиска с использованием фильтров. Кроме того, был описан формат кеширования данных и хранения логов.

В качестве реляционной СУБД была выбрана PostgreSQL, в качестве кеширующей СУБД Redis. Приложение к базе данных было написано на языке C# в формате Web API. Кроме того, в качестве вспомогательного сервиса к нему было написано приложение на языке Python, занимающееся обработкой лингвистических данных. Разработанная система логирования была реализована. Была протестирована ролевая модель.

В результате замеров времени и нагрузочного тестирования было установлено, что использование кеширующей СУБД сокращает время повторного получения данных на 25%. Однако, в связи с необходимостью записи данных в кеширующую СУБД, время получения ответа на новый запрос увеличивается.

Итак, решены следующие задачи:

— проанализирована предметная область параллельных корпусов текстов,

сформированы требования для хранения разметки текстов;

— проанализированы существующие СУБД;
— спроектирована БД для хранения и пополнения параллельного корпуса текстов и ролевая модель;
— спроектировано кеширование запросов с использованием дополнительной БД, спроектирована систему логирования действий пользователей;
— выбраны средства реализации системы, реализована спроектированная БД и необходимый интерфейс для взаимодействия с ней;
— реализована система логирования действий пользователей и проведено тестирование разделения ролей;
— проведено нагрузочное тестирование, определена зависимость времени исполнения запросов от использования кеширующей БД.

Таким образом, цель данной курсовой работы выполнена: разработана база данных для хранения и обработки параллельного корпуса переведённых текстов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Massé, M. REST API Design Cookbook / M. Massé. – Sebastopol: O'Reilly, 2012. – 114 p.

2. Tiwari, L., Pandey, V.G. A Secure Pre-threaded and Pre-forked Unix Client-Server Design for Efficient Handling of Multiple Clients. – 2012. – 5 p.

3. Comparing and Evaluating epoll, select, and poll Event Mechanisms / L. Gammo [et al.] // Proceedings of the 6th Annual Ottawa Linux Symposium. – 2004. – P. 215-225.

4. Ибатова, А.Ш. Определение и понятие лексемы в лексикологии / А.Ш. Ибатова // Science and world. – 2020. – P. 42-43.

5. Rura, L., Vandeweghe, W., Montero Perez, M. Designing a parallel corpus as multifunctional translator's aid // Proceedings of the 18th FIT world Congress. – 2008. – 11 p.

6. Мальцева, М.С. К определению термина «Лингвистический корпус» и его вариантов / М.С. Мальцева // Социально-экономические явления и процессы. – 2011. – №7. – С. 255-258.

7. Khosla S., Acharya H. A survey report on the existing methods of building a parallel corpus // International Journal of Advanced Research in Computer Science. – Vol.9, №4. – 2018. – P. 13-19.

8. Essential Speech and Language Technology for Dutch: Results by the STEVIN programme / P. Spyns [et al.] – The Hague: Springer, 2013. – 414 p.

9. Szudarski P. Corpus Linguistics for Vocabulary: A Guide for Research / P. Szudarski. – London: Routledge, 2017. – 238 p.

10. The Routledge Handbook of Corpus Linguistics / A. O'Keeffe [et al.] – London: Routledge, 2010. – 712 p.

11. Виды разметки – Национальный корпус русского языка. Режим доступа: https://ruscorpora.ru/page/annotation/ (дата обращения: 07.03.2023)

12. Leech G. Corpus Annotation Schemes // Literary and Linguistic Computing. – Vol. 8, №4. – 1993. – P. 275-281.

13. Archer D., Culpeper J., Davies M. Pragmatic Annotation // Corpus Linguistics: An International Handbook. – Berlin: Mouton de Gruyter, 2008. – P. 613-641.

14. Национальный корпус русского языка. Режим доступа: https://ruscorpora.ru/ (дата обращения: 07.03.2023)

15. Poibeau, T. Machine Translation / T. Poibeau. – Cambridge: the MIT Press, 2017. – 215 p.

16. Extensible Markup Language (XML). Режим доступа: https://www.w3.org/XML/ (дата обращения: 01.03.2023)

17. HTML Standard. Режим доступа: https://html.spec.whatwg.org/multipage/ (дата обращения: 01.03.2023)

18. Pęzik P., Ogrodniczuk M., Przepiórkowski A. Parallel and spoken corpora in an open repository of Polish language resources. – 2011. – 6 p.

19. Тао, Ю. Захаров, В.П. Разработка и использование параллельного корпуса русского и китайского языков // Информационные процессы и системы. – 2015.– №4. – С. 18-29.

20. Павлов, М.Н. Подходы к организации загрузки информации из xml-документов в реляционные базы данных // Известия Орловского Государственного Технического Университета. Серия: Информационные системы и технологии. – 2004. – №5(6). – С. 106-109

21. Кренке, Д. Теория и практика построения баз данных / Д. Кренке. – 9-е изд. – СПб.: Питер, 2005. – 859 с.

22. Карпова, И.П. Базы Данных. Учебное пособие / И. П. Карпова. – М.: Московский государственный институт электроники и математики (Технический университет), 2009. – 131 с.

23. Гущин, А.Н. Базы данных: учебно-методическое пособие / А. Н. Гущин. – 2-е изд., испр. и доп. – М.-Берлин: Директ-Медиа, 2015. – 311 с.

24. Аврунев, О.В., Стасышин, В.М. Модели баз данных: учебное пособие / О.Е. Аврунев, В.М. Стасышин. – Новосибирск: Изд-во НГТУ, 2018. – 124 с.

25. Шилин, А.С. Практическая нецелесообразность нормальных форм высокого порядка // Информатика и прикладная математика. – 2016. – №. 22. – С. 116-122.

26. Парфенов, Ю.П. Постреляционные хранилища данных : учеб. пособие / Ю.П. Парфенов. – Екатеринбург: Изд-во Урал. ун-та, 2016. – 120 с.

27. Reverso Context: использование контекстного переводчика. Режим доступа: https://context.reverso.net/перевод/about/ (дата обращения: 10.03.2023)

28. ABBYY Lingvo Live – онлайн-словарь. Режим доступа: https://www.lingvolive.com/ru-ru/ (дата обращения: 10.03.2023)

29. Linguee Dictionary. Режим доступа: https://www.linguee.com/ (дата обращения: 10.03.2023)

30. PostgreSQL: Documentation. Режим доступа: https://www.postgresql.org/docs/ (дата обращения: 15.04.2023)

31. Taipalus, T. Database Management System Performance Comparisons: A Systematic Survey / T. Taipalus. – Jyväskylä: University of Jyväskylä, 2023. – 32 p.

32. MySQL Documentation. Режим доступа: https://dev.mysql.com/doc/ (дата обращения: 15.04.2023)

33. SQL Server technical documentation. Режим доступа: https://learn.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver16 (дата обращения: 15.04.2023)

34. CREATE FUNCTION Statement for Loadable Functions. Режим доступа: https://dev.mysql.com/doc/refman/5.7/en/create-function-loadable.html (дата обращения: 15.04.2023)

35. TIOBE Index for April 2023. Режим доступа: https://www.tiobe.com/tiobe-index/ (дата обращения: 15.04.2023)

36. Документация по C#. Режим доступа: https://learn.microsoft.com/ru-ru/dotnet/csharp/ (дата обращения: 15.04.2023)

37. Python documentation. Режим доступа: https://docs.python.org/3/ (дата обращения: 15.04.2023)

38. Rider: Fast & powerful cross-platform .NET IDE. Режим доступа: https://www.jetbrains.com/rider/ (дата обращения: 15.04.2023)

39. PyCharm: IDE для профессиональной разработки на Python. Режим доступа: https://www.jetbrains.com/ru-ru/pycharm/ (дата обращения: 15.04.2023)

40. Introduction to Redis. Режим доступа: https://redis.io/docs/about/ (дата обращения: 15.04.2023)

41. Create web APIs with ASP.NET Core. Режим доступа: https://learn.microsoft.com/en-us/aspnet/core/web-api/?WT.mc_id=dotnet-35129-website&view=aspnetcore-7.0 (дата обращения: 15.04.2023)

42. Swagger Documentation. Режим доступа: https://swagger.io/docs/ (дата обращения: 15.04.2023)

43. NLog. Flexible & free open-source logging for .NET. Режим доступа: https://nlog-project.org/ (дата обращения: 15.04.2023)

44. Docker Docs: How to build, share and run applications. Режим доступа: https://docs.docker.com/ (дата обращения: 15.04.2023)

45. Wilkins, P. Logging in Action. With Fluentd, Kubernetes and more / Phil Whilkins. – Shelter Island: Manning, 2022. – 394 p.

46. Windows. Режим доступа: https://www.microsoft.com/ru-ru/windows (дата обращения: 15.04.2023)

47. Процессор Intel® Core™ i5-8265U. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/149088/intel-core-i58265u-processor-6m-cache-up-to-3-90-ghz.html (дата обращения: 15.04.2023)

48. Apache JMeter - Apache JMeter™. Режим доступа: https://jmeter.apache.org/ (дата обращения: 25.04.2023)

ПРИЛОЖЕНИЕ А

В листингах **??** – **??** приведены сценарии создания разработанных хранимых функций.

ПРИЛОЖЕНИЕ Б

Презентация к курсовой работе

Презентация содержит 17 слайдов.