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:

«Система хранения конфигураций нейронных сетей»

Студент ИУ7-63Б _____ Г. А. Миронов
(Группа) (Подпись, дата) (И. О. Фамилия)

Руководитель курсовой работы _____ К. Л. Тассов
(Подпись, дата) (И. О. Фамилия)

2022 г.

РЕФЕРАТ

Расчетно-пояснительная записка 69 с., 17 рис., 15 табл., 0 источн., 7 прил.

Ключевые слова: Базы Данных, SQL, NoSQL, Кэширование данных, Нейронные сети, Конфигурация нейронной сети

Объектом разработки является базы данных для хранения конфигураций нейронных сетей.

Цель работы — спроектировать и разработать базу данных для хранения конфигураций нейронных сетей.

Для достижения данной цели необходимо решить следующие задачи:

- проанализировать варианты представления данных и выбрать подходящий вариант для решения задачи;
- проанализировать системы управления базами данных и выбрать подходящую систему для хранения данных.
- спроектировать базу данных, описать ее сущности и связи;
- реализовать интерфейс для доступа к базе данных;
- реализовать программное обеспечение, позволяющее взаимодействовать со спроектированной базой данных.

В результате выполнения работы была спроектирована и разработана база данных для хранения конфигураций нейронных сетей.

По результатам экспериментальных измерений, использование кэширования при получении информации из базы данных позволяет снизить времени отклика системы вплоть до 39 раз, при условии, что запрашиваемая информация находится в кэше.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящее время популярность алгоритмов машинного обучения продолжает неуклонно расти, в связи с чем все чаще возникает проблема хранения результатов работы специалистов в области машинного обучения, в частности - конфигураций нейронных сетей. [**popularity**]

Разработка системы хранения конфигураций нейронных позволит ускорить процесс работы специалистов: такая система гарантирует сохранность полученных ранее результатов работы, а главное — позволяет централизованно взаимодействовать с результатами работы сразу нескольким людям.

Такая система наиболее актуальна для небольших команд разработчиков, использующих в своей деятельности нейронные сети, позволяет хранить конфигурацию любой нейронной сети, в связи с чем является универсальным решением.

Целью данной работы является проектирование и разработка базы данных для хранения конфигураций нейронных сетей.

Для достижения данной цели необходимо решить следующие задачи:

- проанализировать варианты представления данных и выбрать подходящий вариант для решения задачи;
- проанализировать системы управления базами данных и выбрать подходящую систему для хранения данных.
- спроектировать базу данных, описать ее сущности и связи;
- реализовать интерфейс для доступа к базе данных;
- реализовать программное обеспечение, позволяющее взаимодействовать со спроектированной базой данных.

1 Аналитическая часть

В данном разделе проведена формализация задачи: рассмотрено понятие нейронной сети, описан способ представления искуственного нейрона, а так же структура конфигурации нейронной сети. Так же, проведена формализация данных, приведёно сравнение существующих способов хранения данных. Рассмотрены системы управления базами данных, оптимальные для решения поставленной задачи. Описаны проблемы кэширования данных и предложены методы их решения.

1.1 Формализация задачи

Нейронные сети являются подмножеством машинного обучения и лежат в основе алгоритмов глубокого обучения. Их название и структура вдохновлены человеческим мозгом, а алгоритм работы основывается на способе, которым биологические нейроны передают сигналы друг другу. [**neural**]

Простейшая нейронная сеть включает в себя входной слой, выходной (или целевой) слой и, между ними, скрытый слой. Слои соединены через узлы (искусственные нейроны). Эти соединения образуют «сеть» – нейронную сеть – из взаимосвязанных узлов. [**sas-neural**] Пример приведен на Рисунке 1.1.

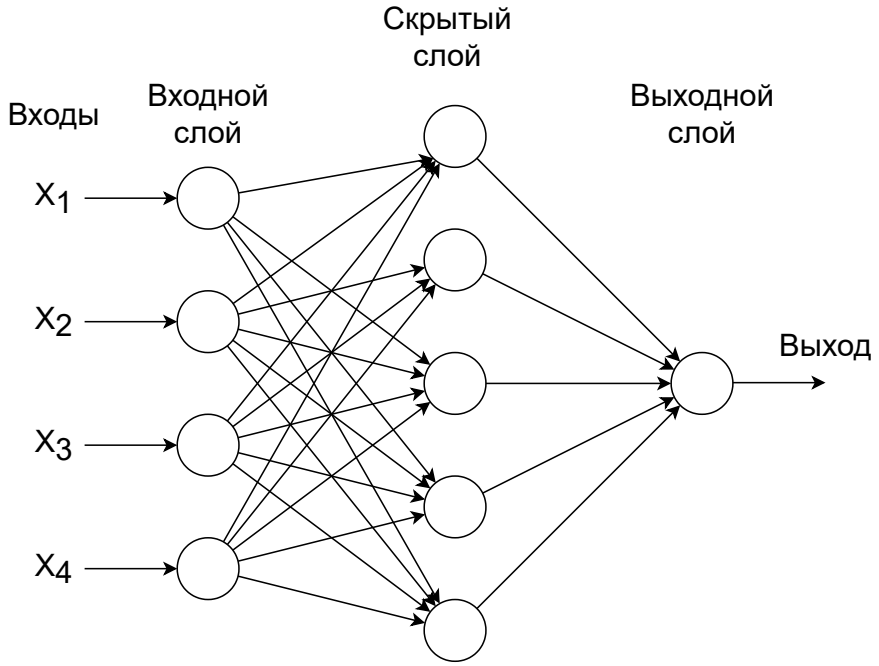

Рисунок 1.1 – Пример схемы нейронной сети

В общем случае искусственный нейрон можно образом, приведенном на

Рисунке 1.2.

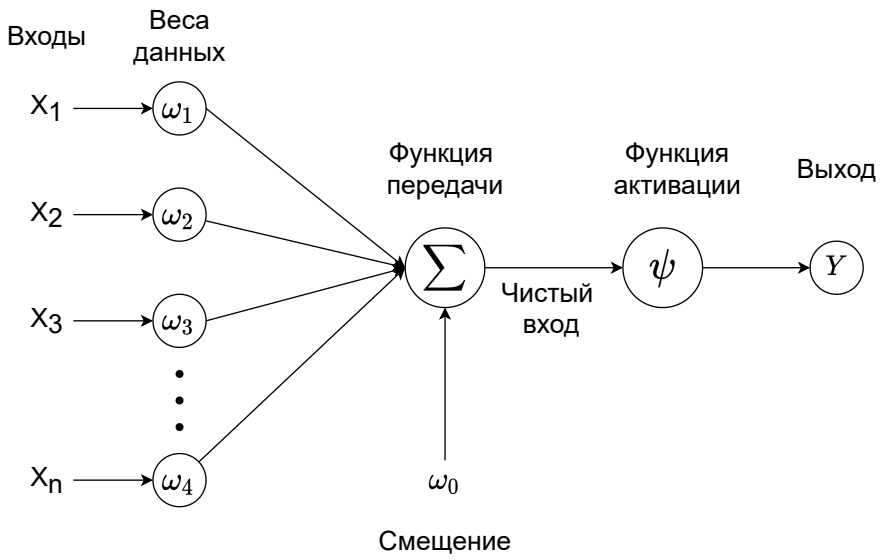

Рисунок 1.2 – Общая схема искусственного нейрона

Данное представление применимо к любому виду нейронных сетей – вне зависимости от типа, нейронные сети реализуются путем упорядочивания нейронов в слои и последующим связыванием соответствующих слоев между собой. [**sharkawy-neural**]

В процессе обучения нейронной сети используется так называемая обучающая выборка – заранее подготовленный набор данных, отражающий суть рассматриваемой предметной области [**sharkawy-neural**]. В зависимости от содержимого обучающей выборки результирующие весовые конфигурации нейронной сети (т. е. веса связей между нейронами, а так же смещения отдельно взятых нейронов) могут отличаться [**mit-neural**]. В связи чем этим одна и та же структура нейронной сети может переиспользована для работы с различными предметными областями.

1.2 Варианты использования

В рамках данной курсовой работы выделены следующие роли пользователей системы:

- пользователь — роль, позволяющая загружать, получать, изменять и удалять существующие конфигурации нейронных сетей;
- аналитик — роль, позволяющая получать статистику работы системы – количество загрузок, изменений конфигураций нейронных сетей и т. д.;

- администратор — роль, позволяющая изменять, удалять любую конфигурацию нейронной сети, блокировать других пользователей, удалять существующих пользователей.

Варианты использования, предусмотренные в рамках данной курсовой работы приведены на Рисунке 1.3:

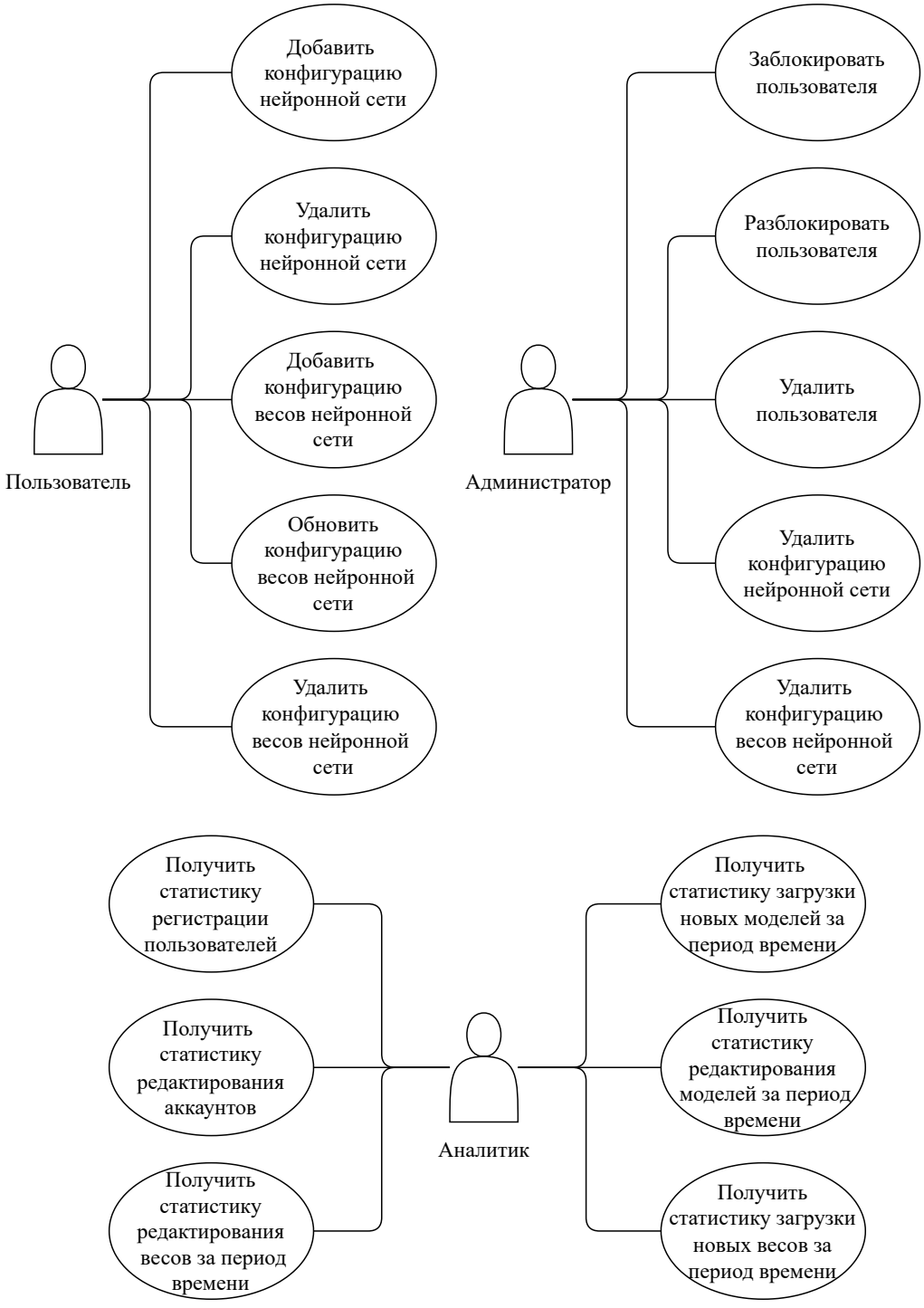

Рисунок 1.3 – Диаграмма вариантов использования

1.3 Формализация данных

Конфигурация нейронной сети включает в себя информацию о [**sharkawy-neural**]:

- отдельных слоях:
  - функции активации конкретных слоев;
  - функции передачи конкретных слоев;
- отдельных нейронах:
  - связях между нейронами;
  - принадлежности конкретных нейронов тому или иному слою;
- результирующих значениях весов:
  - смещения конкретных нейронов;
  - веса связей между конкретными нейронами.

Отдельно стоит отметить, что:

- структура нейронной сети представляет собой, в общем случае, ориентированный граф с n-входами и m-выходами;
- обучающие выборки не являются обязательным компонентом конфигурации нейронной сети.

Результирующая ER-модель представлена на Рисунке 1.4.

Таким образом, система хранения конфигураций нейронных систем должна обеспечивать возможность хранения всей приведенной информации.

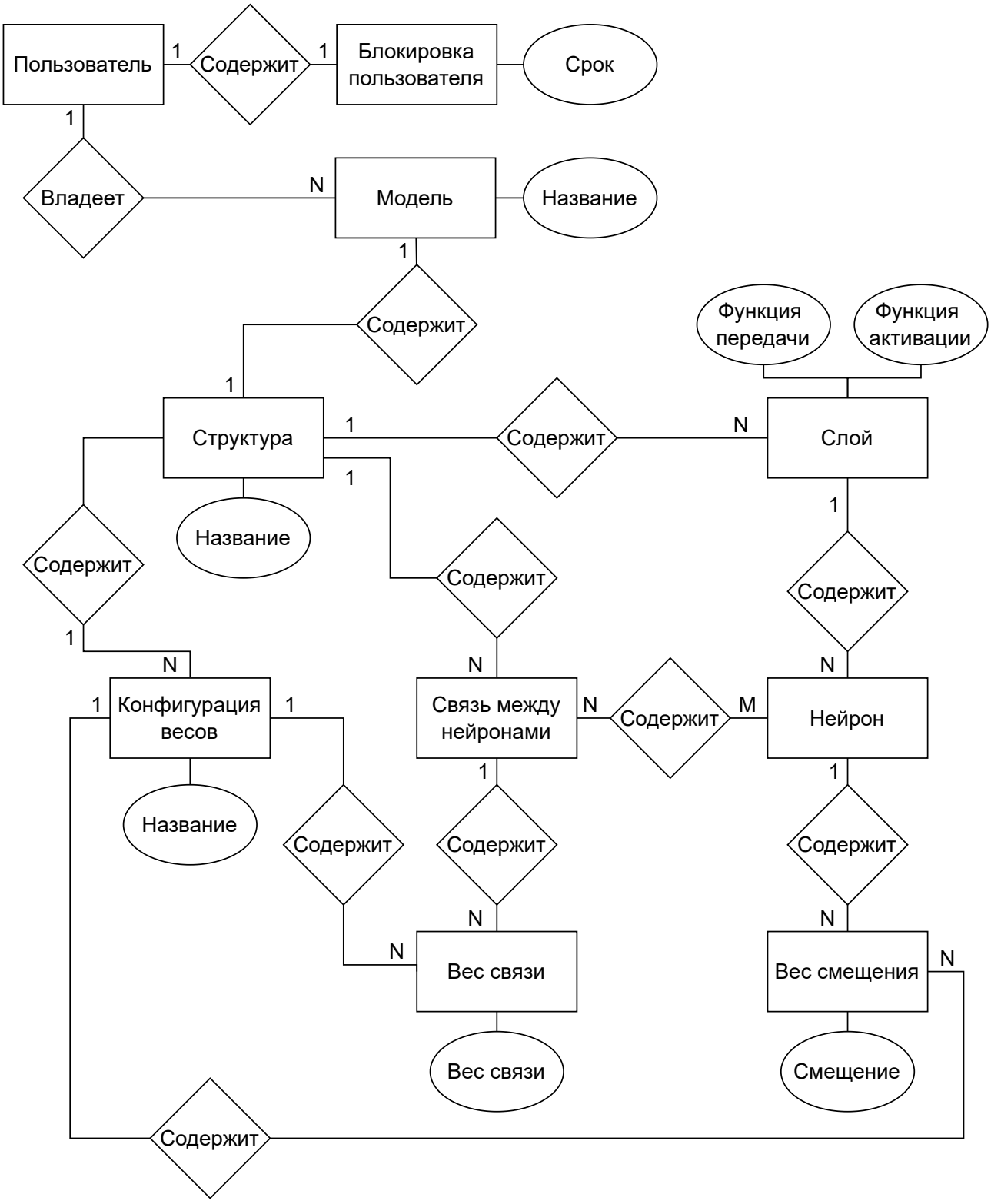

Рисунок 1.4 – er-модель

1.4 Базы данных и системы управления базами данных

База данных – это упорядоченный набор структурированной информации или данных, которые обычно хранятся в электронном виде в компью-

терной системе [**database**]. Для управления базами данных используется системы управления базами данных (далее СУБД).

СУБД – это комплекс программно-языковых средств, позволяющих создать базы данных и управлять данными. Иными словами, СУБД — это набор программ, позволяющий организовывать, контролировать и администрировать базы данных. [**subd**]

1.4.1 Классификация баз данных по способу хранения

Существует множество типов баз данных [**db-comparison**]:

- дореляционные:
  - плоские;
  - иерархические;
  - сетевые;
- реляционные;
- постреляционные:
  - `NoSQL`:
    - «ключ-значение»;
    - документо-ориентированные;
    - графовые;
    - колоночные;
    - временных рядов;
  - NewSQL

В настоящее время наиболее популярными являются реляционные базы данных, а так же различные `NoSQL` решения [**modern-db-comparison**].

Так как любая нейронная сеть может быть представлена описанным выше способом, для хранения конфигурации нейронной сети целесообразно использовать реляционные или графовые базы данных.

Реляционные базы данных

Реляционные базы данных, появившиеся в 1970-х, являются наиболее распространенным типом баз данных. Так, в настоящее время 7 из 10 наиболее популярных баз данных являются реляционными [**db-ranking**]. Кроме того, в реляционных базах используется стандартизированные язык описания и управления данными - `SQL`

В реляционных базах данные представлены в виде строк в таблицах и связей между конкретными строками, в связи с чем такие базы данных подходят для хранения строго структурированной и типизированной информации, причем их можно разделить на 2 группы:

- строковые – используются в системах, в которых предполагается больше количество операций вставки и обновления данных, использующих транзакции (англ. OLTP [**oltp**]).

- колоночные – используются в системах, в которых предполагается большое количество операций выборки данных, в том числе посредством сложных запросов со множественными объединениями таблиц, для анализа хранящейся информации (англ. OLAP [**olap**]).

Реляционные базы данных предоставляют механизм транзакций и удовлетворяют принципам `ACID` [**acid**], в связи с чем, несмотря на сложность внесения изменений в уже созданную базу данных, получили широкое распространение во многих сферах. [**modern-db-comparison**]

Графовые базы данных

Графовые базы данных — это техническая реализация теории графов, концепции, которая была введена в прикладную математику около 200 лет назад [**graphdatabases**].

Такие базы данных представляют данные в виде узлов графа и связей между ними, уделяя особое внимание связям между данными. Благодаря своей структуре, не требуют множественных объединений таблиц, свойственных реляционным базам данных и предоставляют гораздо более гибкую модель для хранения данных [**graphdatabases**].

В настоящее время существуют 2 основных типа графовых баз данных, основное отличие которых заключается в использующихся моделях данных, которые описывают:

- ресурсы (англ. Resource Descriptive Framework), далее RDF;
- свойства ресурсов (англ. Property Graph Databases), далее PGD.

RDF-базы фокусируются на связях между узлами графа. В таких базах данных узлы не содержат никакой информации, а хранимые данные сосредоточены в связях между ними. Такие базы данных наиболее эффективны для публикации данных и обмена информацией. [**graph-vs-rel**]

PGD-базы фокусируются на логической модели данных, пытаясь реализовать наиболее эффективное по времени поиска и объему, требующейся для хранения памяти, представление данных. [**graph-vs-rel**]

Однако, что свойственно нереляционным базам данных, RDF и PGD, в большинстве случаев, не обладают механизмом транзакций, не удовлетворяют принципам ACID, а так же не имеют стандартного интерфейса для описания и управления данными. [**rel-vs-nosql**]

1.4.2 Выбор типа базы данных для решения задачи

База данных для решения задачи должна обладать следующими свойствами:

- наличие строго типизированного языка описания структуры данных и связей между ними (англ. DDL);
- наличие строго типизированного языка управления данными и связями между ними (англ. DML);
- поддержка принципов ACID, транзакций;

В связи с вышеуказанными особенностями конфигураций нейронных сетей, а так же результатами сравнения, приведенными в Таблице 1.1, наиболее предпочтительными являются реляционные базы данных.

Так как разрабатываемая система предполагает большое количество транзакций с операциями вставки, удаления и редактирования данных.

Таблица 1.1 – Сравнение типов баз данных

Критерий	**Реляционные**	**Графовые**
DDL	+	−
DML	+	±
ACID	+	±

1.5 Кэширование данных

В современном мире все больше и больше людей подключаются к Интернету. На текущий момент более 65% населения земного шара имеют подключение к Интернету [**internetusers**].

Учитывая все возрастающее число пользователей сети Интернет, для повышения быстродействия программного обеспечения, используется метод кэширования данных — данные, которые, предположительно, не будут изменены в ближайшее время, сохраняются в независимое хранилище данных. В качестве такого хранилища, обычно, используют NoSQL [**nosql**] in-memory базы данных. Такие базы данных хранят данные в оперативной памяти, что повышает скорость доступа к данным за счет отсутствия затрат на чтение данных со вторичных носителей информации.

1.5.1 Проблемы кэширования данных

Синхронизация данных

Данную проблему можно решить двумя путями:

- реализации на уровне базы данных реализуемого программного продукта триггеров, осуществляющих синхронизацию данных между базой данных и кэшем при изменении / удалении данных из базы;
- реализация на уровне разрабатываемого приложения функций, обеспечивающих синхронизацию данных между базой данных и кэшем при изменении / удалении данных из базы.

Проблема «холодного старта»

При первоначальном запуске системы кэш пуст, в связи с чем все запросы будут напрямую обрабатываться базой данных до тех пор, пока кэш не будет «разогрет» в достаточной мере, чтобы снизить нагрузку на базу данных.

У этой проблемы есть множество решений. Далее приведены два наиболее популярных:

- использование базы данных с журналированием всех операций – при перезагрузке можно восстановить предыдущее состояние кэша с помощью журнала событий, который хранится на диске.

- использование кэша с поддержкой «снимков» хранилища – при перезагрузке можно будет сразу восстановить наиболее актуальное состояние кеша до перезагрузки с помощью «снимка», хранящегося на диске [**tarantool-snapshots**].

В обоих случаях может потребоваться синхронизация кэша и базы данных, так как за время перезагрузки кеша данные, находящиеся в нем могли перестать быть актуальными.

Вывод

В данном разделе была проведена формализация задачи: рассмотрено понятие нейронной сети, описан способ представления искуственного нейрона, а так же структура конфигурации нейронной сети. Так же, была проведена формализация данных, приведёно сравнение существующих способов хранения данных. Рассмотрены системы управления базами данных, оптимальные для решения поставленной задачи. Описаны проблемы кэширования данных и предложены методы их решения.

2 Конструкторская часть

В данном разделе будут спроектированы и приведены отношения таких сущностей разрабатываемой системы, как база данных конфигураций нейронных сетей и база данных кэширования, описан процесс репликации данных. Так же, будет проведено проектирование базы данных для хранения конфигурации нейронной сети на основе реляционной базы данных и проектирование базы данных для реализации кэширования данных на основе NoSQL in-memory базы данных.

2.1 Проектирование отношений сущностей

Так как в проектируемой базе данных наибольший объем будет занимать информация о конфигурации нейронной сети, т.е. о ее структуре и связанными с ней конфигурациями весов, то сохранять в кэшировать следует именно эту информацию — это позволит добиться наиболее заметного снижения времени ответа разрабатываемого приложения [**cache-what-is-it**].

На Рисунке 2.1 представлена схема сущностей, необходимых для реализации разрабатываемого приложения

Рисунок 2.1 – ER-модель разрабатываемой базы данных

2.2 Проектирование базы данных конфигураций нейронных сетей

Система хранения будет реализована с использованием реляционной базы данных, в связи с этим проектирование базы выполняется в виде таблиц и связей между ними.

В разрабатываемой базе данных будет таблиц 10 таблиц, общая er-модель базы данных представлена на Рисунке 2.2.

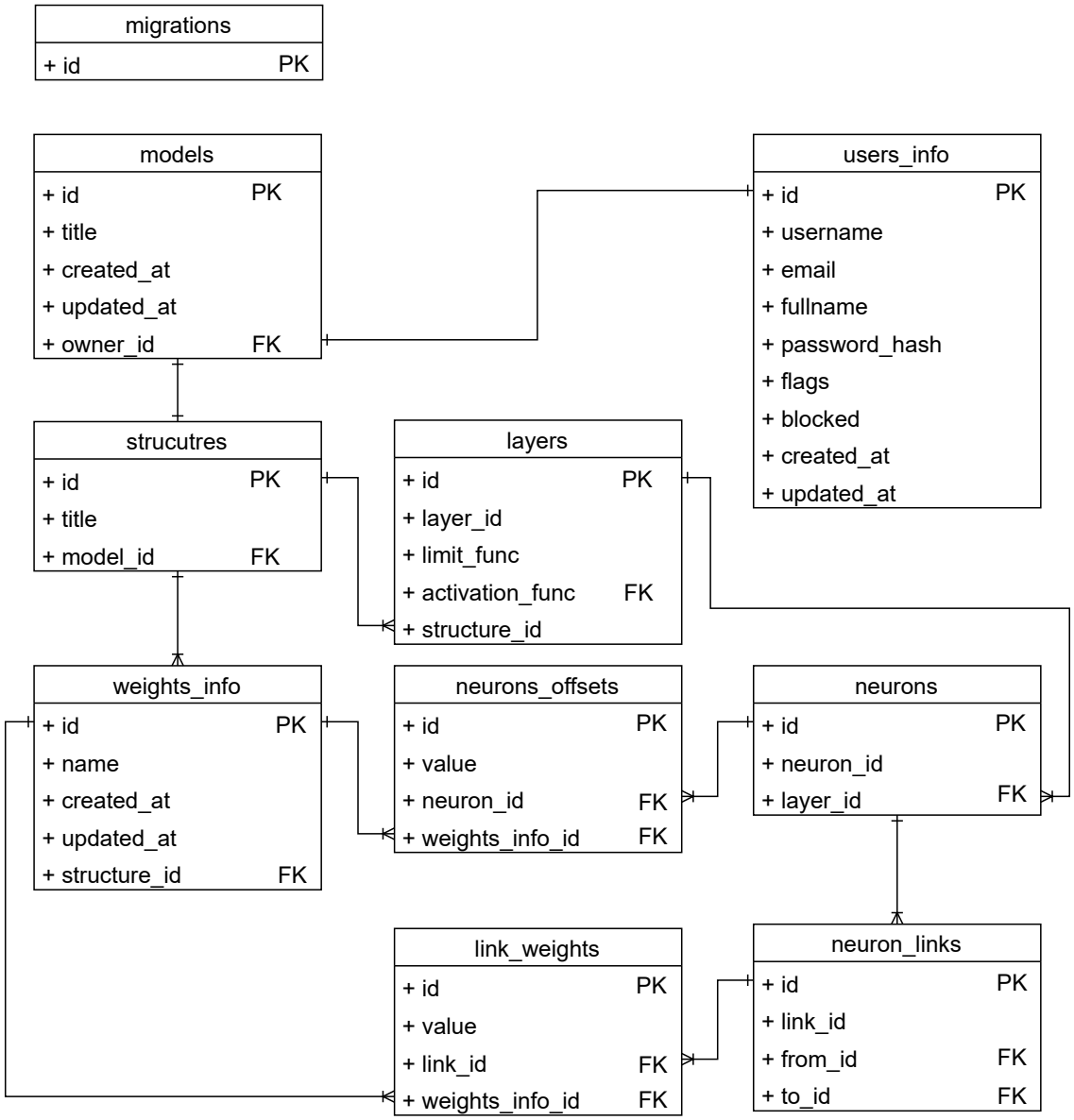

Рисунок 2.2 – ER-модель разрабатываемой базы данных

Рассмотрим отдельно значения полей каждой из таблиц:

Таблица `migrations` содержит всего одно поле:

Таблица 2.1 – Описание полей таблицы `migrations`

Поле	**Значение**
`id`	Уникальный идентификатор завершенной миграции базы данных

Данная таблица используется для фиксаций изменения структуры базы данных — фиксации завершенных миграций базы данных [**database-migrations**].

В таблице `users_info` хранится информация о пользователях системы, а именно:

Таблица 2.2 – Описание полей таблицы `users_info`

Поле	**Значение**
`id`	Уникальный идентификатор пользователя в системе. Этот идентификатор используется для связи владельца конфигурации нейронной сети с, собственно, конфигурации, загруженной в систему — удалить данные из системы может только пользователь, загрузивший их
`username`	Имя пользователя для отображения в системе
`email`	Адрес электронной почты пользователя
`fullname`	Полное имя пользователя
`password_hash`	Хеш пароля пользователя
`flags`	Значения этого поля определяют допустимые действия в системе
`blocked`	Время завершения блокировки пользователя — до этого момента доступ пользователя к системе ограничен
`created_at`	Время регистрации пользователя
`updated_at`	Время последнего редактирования информации о пользователе

В Таблице `models` представлена информация о загруженных нейронных сетях, а именно:

Таблица 2.3 – Описание полей таблицы `models`

Поле	**Значение**
`id`	Уникальный идентификатор конфигурации
`title`	Имя пользователя для отображения в системе
`created_at`	Время регистрации пользователя
`updated_at`	Время последнего редактирования информации о пользователе
`owner_id`	Идентификатор пользователя, загрузившего данную конфигурацию

Такой способ представления нейронной сети позволяет получить всю доступную информацию о ней, зная лишь ее уникальный идентификатор.

Данная таблица имеет отношение многие-к-одному с таблицей `users_info` — у одного пользователя может иметься множество конфигураций нейронных сетей.

В таблице `structures` представлена информация о стуктуре нейронной сети, а именно:

Таблица 2.4 – Описание полей таблицы `structures`

Поле	**Значение**
`id`	Уникальный идентификатор структуры
`title`	Название структуры нейронной сети
`model_id`	Идентификатор модели, структура которой описывается данной записью в таблице

В таблице `layers` представлена информация о слоях нейронной сети, а именно:

Таблица 2.5 – Описание полей таблицы `layers`

Поле	**Значение**
`id`	Уникальный идентификатор слоя
`layer_id`	Идентификатор слоя, полученный из входных данных
`limit_func`	Идентификатор функции передачи (Конкретная функция в базе не хранится из-за проблем совместимости с различными реализациями нейронных сетей)
`activation_func`	Идентификатор функции активации (Конкретная функция в базе не хранится из-за проблем совместимости с различными реализациями нейронных сетей)
`structure_id`	Идентификатор структуры, которая включает в себя слой, описываемый данной записью в таблице

В таблице `neurons` представлена информация о искуственных нейронах, содержащихся в нейронной сети, а именно:

Таблица 2.6 – Описание полей таблицы `neurons`

Поле	**Значение**
`id`	Уникальный идентификатор нейрона
`neuron_id`	Идентификатор нейрона, полученный из входных данных
`layer_id`	Идентификатор слоя нейронной сети, в котором содержится искусственный нейрон, описываемый данной записью в таблице

Таблица `neuronlinks` описывает связи между искуственными нейронами:

Таблица 2.7 – Описание полей таблицы `neuron_links`

Поле	**Значение**
`id`	Уникальный идентификатор связи
`link_id`	Идентификатор связи, полученный из входных данных
`from_id`	Идентификатор нейрона, в котором начинается данная связь
`to_id`	Идентификатор нейрона, в котором завершается данная связь

Далее рассматривается описание конфигурации весов нейронной сети.

Таблица `weightsinfo` описывает конфигурацию весов нейронной сети, а именно:

Таблица `neuron_offsets` описывает вес смещения конкретных нейронов и содержит следующую информацию:

Таблица `link_weights` описывает вес конкретных связей между нейронами и содержит следующую информацию:

2.3 Проектирование базы данных кэширования

База данных кеширования будет реализована посредством NoSQL in-memory базы данных. Как указывалось ранее, способ представления данных в таких базах данных зависит от выбранной базы, в связи с чем проектирование такой базы ограничивается высокоуровневым описанием хранящихся данных.

Таблица 2.8 – Описание полей таблицы `weights_info`

Поле	**Значение**
`id`	Уникальный идентификатор конфигурации весов нейронной сети
`name`	Название данной конфигурации — используется пользователями для идентификации конфигураций
`created_at`	Время добавления данной конфигурации весов
`updated_at`	Время последнего редактирования информации о данной конфигурации весов
`structure_id`	Идентификатор структуры нейронной сети, которая описывается данной конфигурацией весов

Таблица 2.9 – Описание полей таблицы `neuron_offsets`

Поле	**Значение**
`id`	Уникальный идентификатор веса смещения нейрона
`value`	Величина смещения
`neuron_id`	Идентификатор нейрона, смещение которого описывает данная запись в таблице
`weights_info_id`	Идентификатор конфигурации весов нейронной сети, в которую входит данный вес смещения

Таблица 2.10 – Описание полей таблицы `link_weights`

Поле	**Значение**
`id`	Уникальный идентификатор веса связи между нейронами
`value`	Вес связи
`link_id`	Идентификатор связи, вес которой описывает данная запись в таблице
`weights_info_id`	Идентификатор конфигурации весов нейронной сети, в которую входит данный вес смещения

Ранее указывалось, что разрабатываемая база данных кэширования должна предоставлять возможность хранения информации о нейронной сети и конфигурации весов нейронной сети.

Для информации о конфигурации нейронной сети это можно представить образом, представленным в Таблице 2.11

Таблица 2.11 – Описание хранилища `model`

Поле	**Значение**
`id`	Уникальный идентификатор нейронной сети
`info`	Информация о конфигурации нейронной сети

Для информации о конфигурации весов представление аналогично и представлено в Таблице 2.12:

Таблица 2.12 – Описание хранилища `weight`

Поле	**Значение**
`id`	Уникальный идентификатор нейронной сети
`info`	Информация о конфигурации весов нейронной сети

Конкретная реализация данных хранилищ будет определена после выбора NoSQL базы данных для реализации хранилища кэша, однако уже на текущем этапе можно определить следующие требования к итоговой реализации данных хранилищ:

- возможность сохранения информации "как есть" — т.е. отсутствие необходимости в специальной подготовке данных перед сохранением в хранилище;
- хранение информации о конфигурациях в виде, в котором эта информация будет передана пользователю — т.е. информация, хранящаяся в кэше не должна требовать дополнительной обработки после получения ее из хранилища.

2.4 Соблюдение целостности данных

Для соблюдения целостности данных в проектируемой базе данных будут использованы внешние ключи, ограничения, триггеры, а так же — ролевая модель.

Триггеры

На Рисунке 2.3 представлена схема алгоритма триггера, срабатывающего после обновления строки в таблице `models`.

Рисунок 2.3 – Схема алгоритма триггера обновления информации о нейронной сети

На Рисунке 2.4 представлена схема алгоритма триггера, срабатывающего после обновления строки в таблице `weights_info`.

Так же, в проектируемой базе данных должны быть реализованы триггеры, осуществляющие синхронизацию данных между, собственно, базой данных и кэшем.

На Рисунке 2.5 представлена схема алгоритма триггера, удаляющего информацитю о конфигурации нейронной сети из кэша.

Данный тригер должен срабатывать при обновлении или удалении записей из следующих таблиц: `models`, `structures`, `weights_info`.

Рисунок 2.4 – Схема алгоритма триггера обновления информации о конфигурации весов нейронной сети

На Рисунке 2.6 представлена схема алгоритма триггера, удаляющего информацитю о конфигурации весов нейронной сети из кэша.

Данный тригер должен срабатывать при обновлении или удалении записей из следующих таблиц: `models`, `structures`, `weights_info`.

Внешние ключи

В проектируемой базе данных используются следующие внешние ключи:

- для таблицы `models` – внешний ключ `owner_id`, ссылающийся на поле `id` таблицы `users_info`;
- для таблицы `structures` – внешний ключ `model_id`, ссылающийся на поле `id` таблицы `models`;
- для таблицы `layers` – внешний ключ, `strucuure_id`, ссылающийся на поле `id` таблицы `structures`;
- для таблицы `neurons` – внешний ключ, `layer_id`, ссылающийся на поле `id` таблицы `layers`;
- для таблицы `neuron_links`:
  - внешний ключ `from_id`, ссылающийся на поле `id` таблицы `neurons`;
  - внешний ключ `to_id`, ссылающийся на поле `id` таблицы `neurons`;

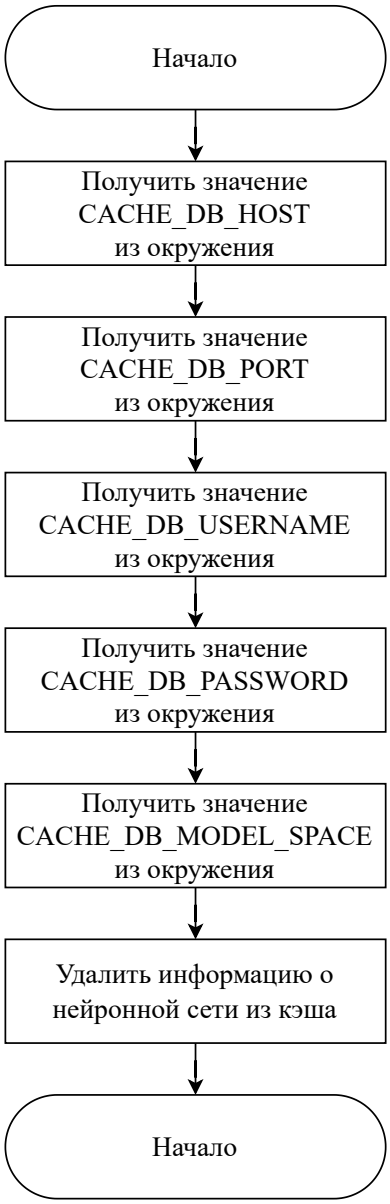

Рисунок 2.5 – Схема алгоритма триггера удаления информации о нейронной сети из кэша

- для таблицы `weights_info` – внешний ключ, `structure_id`, ссылающийся на поле `id` таблицы `structures`;
- для таблицы `neuron_offsets`:
  - внешний ключ `weights_info_id`, ссылающийся на поле `id` таблицы `weights_info`;
  - внешний ключ `neuron_id`, ссылающийся на поле `id` таблицы `neurons`;
- для таблицы `link_weights`:

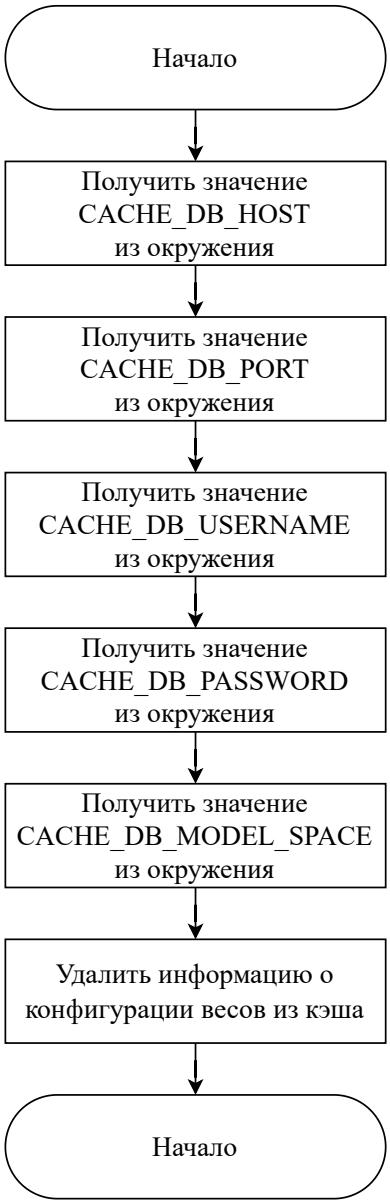

Рисунок 2.6 – Схема алгоритма триггера удаления информации о конфигурации весов нейронной сети из кэша

– внешний ключ `weights_info_id`, ссылающийся на поле `id` таблицы `weights_info`;

– внешний ключ `link_id`, ссылающийся на поле `id` таблицы `neuron_links`;

Ограничения

В проектируемой базе данных будет использоваться одно ограничение: поле `flags` таблицы `user_info` должно быть неотрицательным.

Ролевая модель

Ролевая модель базы данных должна содержать 3 роли:

- обычный пользователь — имеет возможность:
  - добавлять и получать записи из таблицы `users_info`;
  - добавлять, обновлять, получать и удалять записи из таблицы `models`;
- аналитик — имеет возможность получать записи из таблиц `users_info`, `models`, `weights_info`;
- администратор — имеет возможность добавлять, обновлять, получать и удалять записи из любых таблиц.

Вывод

В данном разделе были спроектированы и приведены отношения таких сущностей разрабатываемой системы, как база данных конфигураций нейронных сетей и база данных кэширования, описан процесс репликации данных. Так же, было проведено проектирование базы данных для хранения конфигурации нейронной сети на основе реляционной базы данных и проектирование базы данных для реализации кэширования данных на основе NoSQL in-memory базы данных.

3 Технологическая часть

В данном разделе обосновывается выбор конкретных СУБД для решения задачи, описывается общая архитектура реализуемого программного продукта. Кроме того, приводится описание архитектуры реализуемого приложения: описание архитектурного паттерна, используемого при проектировании, описание конкретных компонентов приложения, а так же - определяются средства реализации приложения, описывается интерфейс взаимодействия с приложением, приводятся детали реализации разрабатываемого приложения.

3.1 Обзор СУБД

В данном подразделе будут рассмотрены популярные СУБД, которые могут быть использованы для реализации хранения в разрабатываемом программном продукте.

Oracle Database

Oracle Database [**oracle**] – объектно-реляционная система управления базами данных разрабатываемая компанией Oracle [**oracle-company**]. На данный момент, рассматриваемая СУБД является наиболее популярной в мире. [**oracle-popular**]

Все транзакции Oracle Database обладают свойствами ACID [**acid**], поддерживает триггеры, внешние ключи и хранимые процедуры. Данная СУБД подходит для разнообразных рабочих нагрузок и может использоваться практически в любых задачах. Особенностью Oracle Database является быстрая работа с большими массивами данных.

MySQL

MySQL [**mysql**] – свободная реляционная система управления базами данных. Разработку и поддержку MySQL осуществляет корпорация Oracle.

Рассматриваемая СУБД имеет два основных движка хранения данных: InnoDB [**innodb**] и myISAM [**myisam**]. Движок InnoDB полностью совместим с принципами ACID, в отличии от движка myISAM. СУБД MySQL подходит для использования при разработке веб-приложений, что объясняется очень

тесной интеграцией с популярными языками PHP [**php**] и Perl [**perl**].

PostgreSQL

PostgreSQL [**postgresql**] – это свободно распространяемая объектно-реляционная система управления базами данных, наиболее развитая из открытых СУБД в мире и являющаяся реальной альтернативой коммерческим базам данных [**postgresql-fact**].

PostgreSQL предоставляет транзакции, обладающие свойствами ACID, автоматически обновляемые представления, материализованные представления, триггеры, внешние ключи и хранимые процедуры. Данная СУБД предназначена для обработки ряда рабочих нагрузок, от отдельных компьютеров до хранилищ данных или веб-сервисов с множеством одновременных пользователей.

3.2 Обзор in-memory NoSQL СУБД

Redis

Redis [**redis**] – резидентная система управлениями базами данных класса NoSQL с открытым исходным кодом. Основной структурой данных, с которой работает Redis является структура типа «ключ-значение». Данная СУБД используется как для хранения данных, так и для реализации кэшей и брокеров сообщений.

Redis хранит данные в оперативной памяти и снабжена механизмом «снимков» и журналирования, что обеспечивает постоянное хранение данных. Существует поддержка репликации данных типа master-slave, транзакций и пакетной обработки команд.

Все данные Redis хранит в виде словаря, в котором ключи связаны со своими значениями. Ключевое отличие Redis от других хранилищ данных заключается в том, что значения этих ключей не ограничиваются строками. Поддерживаются следующие абстрактные типы данных:

- строки;
- списки;
- множества;

- хеш-таблицы;
- упорядоченные множества.

Тип данных значения определяет, какие операции доступные для него; поддерживаются высокоуровневые операции: например, объединение, разность или сортировка наборов.

Tarantool

Tarantool [**tarantool**] – это платформа in-memory вычислений с гибкой схемой хранения данных для эффективного создания высоконагруженных приложений. Включает себя базу данных и сервер приложений на языке программирования Lua [**lua**].

Tarantool обладает высокой скоростью работы по сравнению с традиционными СУБД. При этом, в рассматриваемой платформе для транзакций реализованы свойства ACID, репликация `master-slave` [**master-slave**] и `master-master` [**master-master**], как и в традиционных СУБД.

Для хранения данных используется кортежи (англ. tuple) данных. Кортеж – это массив не типизированных данных. Кортежи объединяются в спейсы (англ. space), аналоги таблицы из реляционной модели хранения данных.

В рассматриваемой СУБД реализованы два движка хранения данных: memtx [**memtx-vinyl**] и vinyl [**memtx-vinyl**]. Первый хранит все данные в оперативной памяти, а второй на диске. Для каждого спейса можно задавать различный движок хранения данных.

Каждый спейс должен быть проиндексирован первичным ключом. Кроме того, поддерживается неограниченное количество вторичных ключей. Каждый из ключей может быть составным.

В Tarantool реализован механизм «снимков» текущего состояния хранилища и журналирования всех операций, что позволяет восстановить состояние базы данных после ее перезагрузки.

3.3 Выбор СУБД для решения задачи

Для решения задачи хранения конфигураций нейронный сетей была выбрана СУБД PostgreSQL, потому что данная СУБД имеет поддержку

языка `plpython3u` [**plpython3u**], который упрощает процесс интеграции базы данных в разрабатываемое приложение. Кроме того, PostgreSQL проста в развертывании.

Для кэширования данных была выбрана СУБД Tarantool, так как она проста в развертывании и переносимости, и имеет подходящие коннекторы для базы данных PostgreSQL [**postgresql-fact**].

3.4 Архитектура приложения

Предполагается, что разрабатываемое приложения является частью некоторого большого сервиса. Доступ к данным, хранящимся в приложении предоставляется посредством `REST API` [**rest**].

Серверная часть приложения взаимодействует с базами данных посредством коннекторов, позволяющих выполнять запросы к базе данных на языке программирования, используемом для разработки приложения.

Общая схема взаимодействия с базой данных представлена на Рисунке 3.1

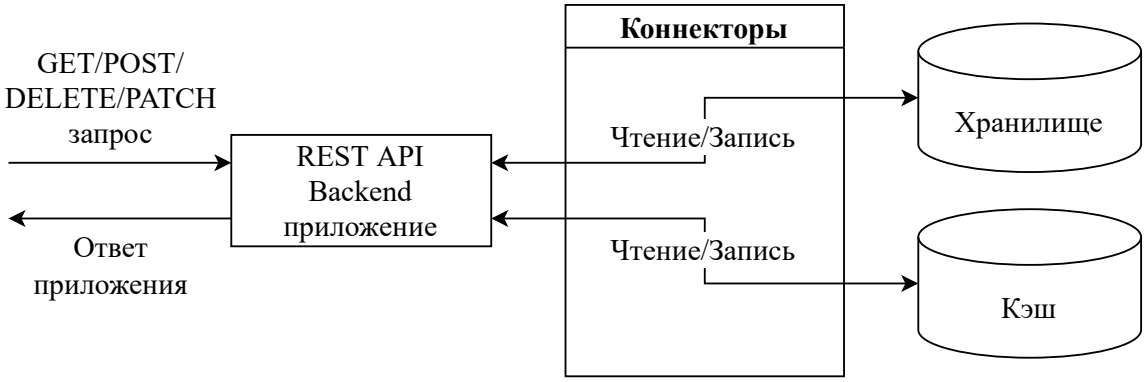

Рисунок 3.1 – Общая схема взаимодействия с базой данных

В качестве основного паттерна проектирования архитектуры приложения будет использован `MVP` [**mvp**]. Разрабатываемое приложение будет состоять из следующих компонентов:

На Рисунке 3.2 представлены следующие компоненты:

- `Main` — основной компонент приложения, содержит точку входа, а так же производит конфигурирование других компонентов приложения;
- `Config Manager` — компонент, предоставляющий данные для конфигурирования приложения;

- `Authorizer` — компонент, реализующий механизмы аутентификации и авторизации в системе [**auth**];
- `View` — компонент, являющийся частью паттерна `MVP` - предоставляет доступ к системе. Так как в данной курсовой работе реализуется доступ посредством `REST API`, то данный компонент должен быть представлен обработчиками `HTTP`-запросов [**http**];
- `Interactors` — компонент, реализующий `Presenter` паттерна;
- `Entites` — компонент, содержащий реализации доменных моделей — `Model` паттерна;
- `Database` — компонент, предоставляющий доступ к базе данных, хранящей конфигурации нейронных сетей;
- `Cache` — компонент, предоставляющий доступ к кэшу приложения.

Соответствующая диаграмма компонентов разрабатываемого приложения приведена на Рисунке 3.2.

3.5 Средства реализации

Разрабатываемое приложение будет являться web-сервером, предоставляющим `REST API`. В связя с этим в качестве языка разработки будет использоваться язык `Golang` [**golang**], так как этот язык создан для разработки различных web-приложений и для него существуют готовые коннекторы к выбранным базам данных.

Для реализации `REST API` был использован фреймворк `Gin Gonic` [**gingonic**].

Для коммуникации серверной части приложения с базами данных были использованы коннекторы `gorm` [**gorm**] для PostgreSQL и `go-tarantool` [**go-tarantool**] для tarantool соответственно.

Для упаковки приложения в готовый продукт была выбрана система контейнеризации `Docker` [**docker**] и система управления развертыванием контейнеров `Docker-Compose` [**docker-compose**]. С помощью `Docker`, можно создать изолированную среду для программного обеспечения, которое можно

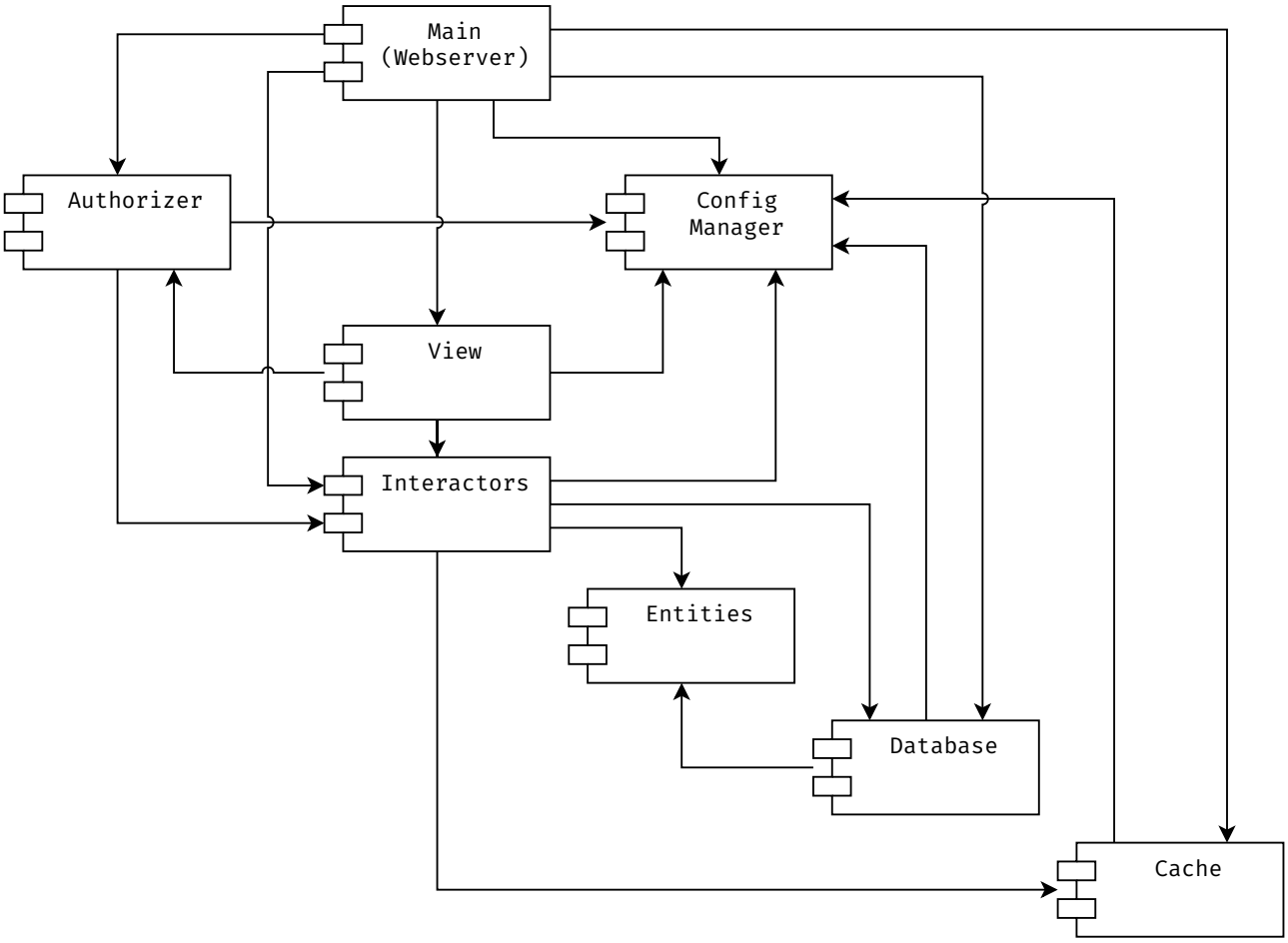

Рисунок 3.2 – Диаграмма компонентов

будет развернуть на различных операционных система без дополнительного вмешательства для обеспечения совместимости.

Тестирование производилось с помощью стандартных средств языка golang — go test, позволяющих проводить как модульные, так и интеграционные тесты.

3.6 Детали реализации

Скрипт создания базы данных, включая создание индексов, органичений, триггеров и описанных выше ролей, приведена в Листингах А.1 — Е.1.

Проектируемое приложение должно реализовывать доступ к разрабатываемой базе данных. Для этого может быть реализован и использован `REST API` [**rest**], т.е. приложение будет являться web-сервером, API для взаимодействия с которым предстоит разработать.

При этом в разрабатываемом API необходимо учесть различные роли пользователей и соответствующий им функционал, описанный диаграммой

вариантов использования и представленный на Рисунке 1.3, и ограничения. `REST API` проектируемого приложения представлен в Таблице 3.1.

Таблица 3.1 – Описание `REST API` реализуемого приложения

Путь	Метод	Описание
/api/v1/registration	POST	Метод для регистрации нового пользователя в системе
/api/v1/login	POST	Метод для авторизации существующего пользователя в системе
/api/v1/refresh	GET	Метод обновления токена доступа
/api/v1/logout	GET	Метод выхода из системы
/api/v1/users	GET	Метод получения информации о пользователях системы
/api/v1/models	POST GET PATCH DELETE	Методы взаимодействия с информацией о конфигурациях нейронных сетей
/api/v1/models/weights	POST GET DELETE PATCH	Методы взаимодействия с информацией о конфигурациях весов нейронных сетей
/api/v1/admin/login	POST	Метод авторизации администратора разработанного приложения
/api/v1/admin/refresh	GET	Метод обновления токена доступа для администратора
/api/v1/admin/logout	GET	Метод выхода из системы для администратора
/api/v1/admin/users	GET DELETE	Методы управления информацией о пользователях для администратора
Продолжение на следующей странице		

Таблица 3.1 – продолжение

Путь	Метод	Описание
/api/v1/admin/users/blocked	GET DELETE PATCH	Методы взаимодействия с информацией о блокировке пользователей
/api/v1/admin/models	GET DELETE	Методы взаимодействия с информацией о конфигурациях нейронных сетей любого пользователя
/api/v1/admin/models/weights	GET DELETE	Методы взаимодействия с информацией о конфигурациях весов нейронных сетей любого пользователя
/api/v1/stat/login	POST	Метод авторизации аналитика в системе
/api/v1/stat/refresh	GET	Метод обновления токена доступа для аналитика
/api/v1/stat/logout	GET	Метод выхода из ситсемы для аналитика
/api/v1/stat/users	GET	Метод получения статистики по пользователям
/api/v1/stat/models	GET	Метод получения статистики по конфигурациям нейронных сетей
/api/v1/stat/weights	GET	Метод получения статистики по конфигурациям весов нейронных сетей
		Конец таблицы

Развертывание приложения выполнялось посредством Docker-контейнеров и системой Docker-compose, файлы конфигурации для которых приведены в Листингах Ж.1–Ж.9.

Вывод

В данном разделе был обоснован выбор конкретных СУБД для решения задачи, описана общая архитектура реализованного программного продукта. Кроме того, было приведено описание архитектуры реализуемого приложения: описан архитектурный паттерн, использованный при проектировании, описаны конкретные компоненты приложения, а так же - определены средства реализации приложения, описан интерфейс взаимодействия с приложением, приведены детали реализации разрабатываемого приложения.

4 Исследовательская часть

В данном разделе представлена постановка эксперимента по сравнению занимаемого времени для получения данных о конфигурации нейронных сетей посредством реализованной базы данных и приложения, обеспечивающего доступ к ней с использованием и без использования кэширования: описан эксперимент, приведены технические характеристики устройств, использованных во время тестирования, приведены результаты измерений, полученные в ходе эксперимента.

4.1 Цель эксперимента

Целью эксперимента является сравнение времени, требуемого для получения данных о конфигурации нейронной сети посредством разработанного приложения с и без использования кэширования данных.

Для достижения этой цели требуется:

- сгенерировать конфигурации нейронных сетей отвечающие требуемым характеристикам'
- загрузить сгенерированные конфигурации в разработанную базу данных;
- произвести экспериментальные замеры времени, необходимые для получения информации о сгенерированных конфигурациях нейронных сетей.

4.2 Описание эксперимента

Сравнить занимаемое время можно при помощи отключения реализованного механизма кэширования. Для этого будет достаточно отключить базу данных, хранящую данные о кэшировании и каждый раз выполнять запрос напрямую к базе данных хранящую информацию о конфигурациях нейронных сетей.

Для проведения будут использоваться разные размеры запрашиваемых конфигураций нейронных сетей. Будут произведены операции, для того чтобы запрашиваемые конфигурации нейронных сетей могли оказаться в кэше.

В поставленном эксперименте одна нейронная сеть будет состоять из 8, 16, 32 ..., 256 слоев по 16, 32, 64, 128 нейронов в каждом. Связи между

нейронами организованы следующим образом: каждый нейрон из j-ого слоя соединяется с каждым нейроном из $j + 1$-ого слоя.

Таким образом, будут рассмотрены конфигурации нейронных сетей с количеством связей между нейронами от 1792, до 4177920 штук.

Пример такой нейронной сети приведен на Рисунке 4.1.

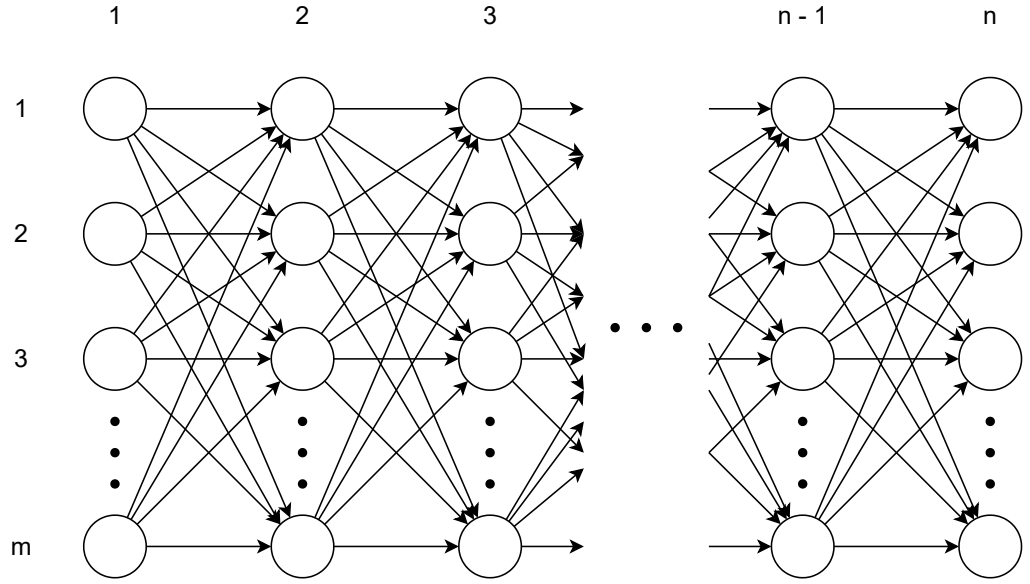

Рисунок 4.1 – Общая структура использующейся конфигурации нейронной сети

4.3 Технические характеристики

Эксперимент производился с использованием двух ЭВМ - сервера и клиента.

В качестве сервера выступал `MacBook Pro 2020`, технические характеристики:

- процессор: Intel Core™ i7-1068NG7 [**i7**] CPU @ 2.30ГГц;
- память: 16 Гб;
- операционная система: `macOS Big Sur` [**bigsur**] 11.6.

В качестве клиента использовался `HP ProBook 440 G5`, технические характеристики:

- процессор: Intel Core™ i5-8250U [**i5**] CPU @ 1.60 Ггц;
- память: 32 Гб;

- операционная система: Manjaro [**manjaro**] Linux [**linux**] 21.2.26 64-bit.

Исследование проводилось на ноутбуках, подключенным к сети электропитания. Во время тестирования ноутбуки были нагружены только встроенными приложениями окружения рабочего стола, окружением рабочего стола, а также непосредственно тестируемой системой.

4.4 Результат эксперимента

Полученные в ходе эксперимента данные, представлены в Таблице 4.1.

Таблица 4.1 – Замеры времени для различных конфигураций, с

Количество нейронов в слое, шт.	**Количество слоев, шт.**							
	16		**32**		**64**		**128**	
	БД	Кэш	БД	Кэш	БД	Кэш	БД	Кэш
8	0.04	0.01	0.05	0.02	0.11	0.027	0.008	0.11
16	0.05	0.02	0.08	0.027	0.21	0.042	0.11	0.10
32	0.06	0.02	0.15	0.03	0.47	0.047	1.81	0.09
64	0.09	0.03	0.21	0.05	0.9	0.08	3.32	0.14
128	0.14	0.05	0.55	0.075	2.05	0.12	6.74	0.26
256	0.27	0.08	1.33	0.1	5.70	0.23	23.66	0.6

Для большей наглядности представим те же данные в виде графиков.

На Рисунке 4.2 представлено сравнение результатов замеров времени для конфигураций с 16 нейронами в каждом слое.

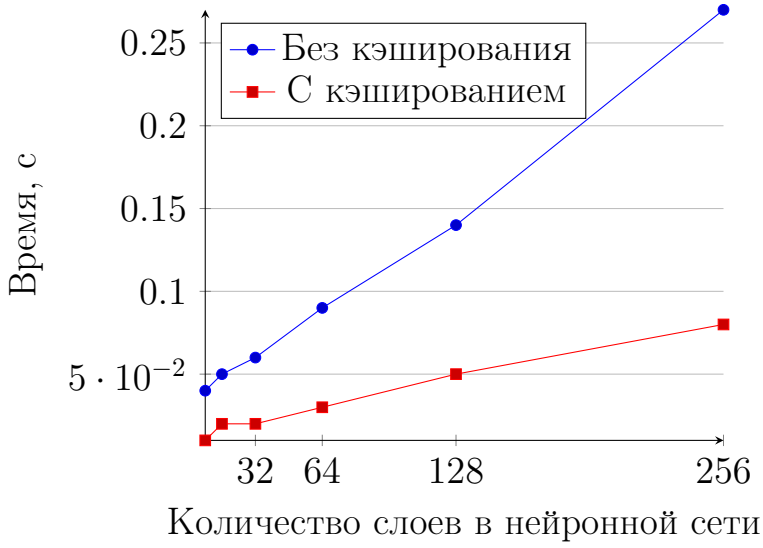

Рисунок 4.2 – Зависимость времени от размера конфигурации нейронной сети (16 нейронов в слое)

На Рисунке 4.3 представлено сравнение результатов замеров времени для конфигураций с 32 нейронами в каждом слое.

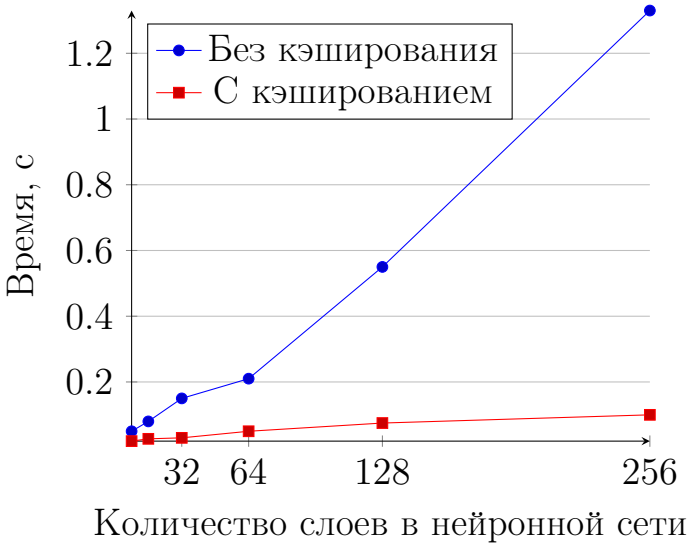

Рисунок 4.3 – Зависимость времени от размера конфигурации нейронной сети (32 нейрона в слое)

На Рисунке 4.4 представлено сравнение результатов замеров времени для конфигураций с 64 нейронами в каждом слое.

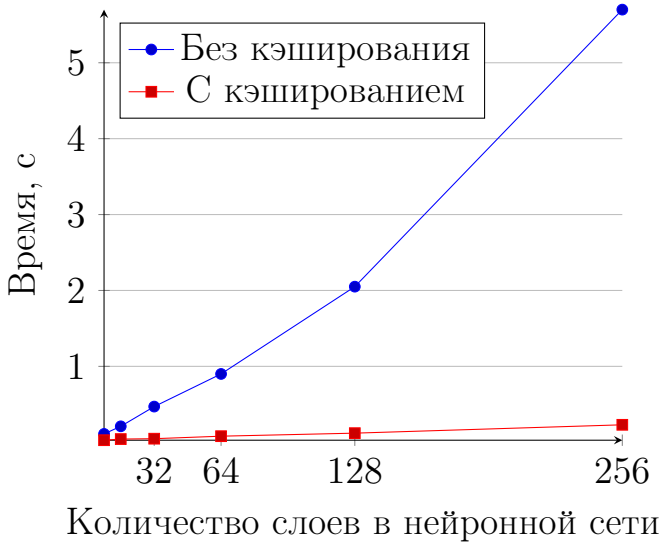

Рисунок 4.4 – Зависимость времени от размера конфигурации нейронной сети (64 нейрона в слое)

На Рисунке 4.5 представлено сравнение результатов замеров времени для конфигураций с 128 нейронами в каждом слое.

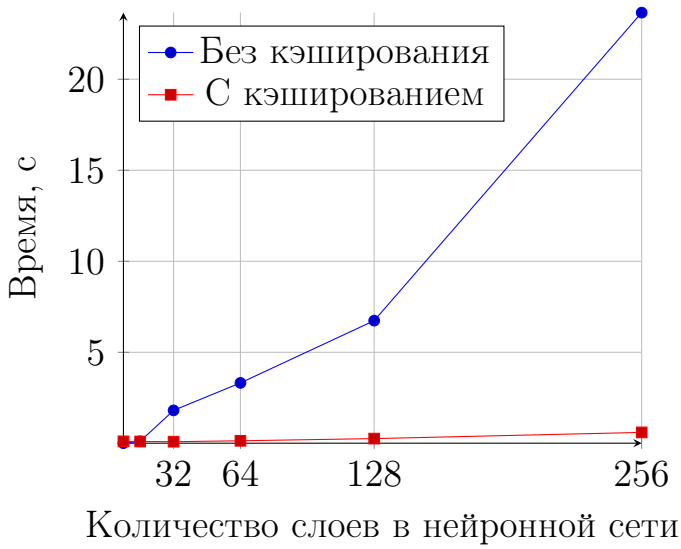

Рисунок 4.5 – Зависимость времени от размера конфигурации нейронной сети (128 нейронов в слое)

Вывод

В результате анализа полученных экспериментально данных о времени, необходимом для получения данных о конфигурации нейронной сети, кэширование данных показало более высокие результаты:

- вне зависимости от размера нейронной сети, приложение с кэшированием всегда оказывалось быстрее;
- при большом размере (256 слоев по 128 нейронов в каждом) нейронной сети, приожение с кэшированием оказалось в 39.4 раза быстрее (при условии, что запрашиваемые данные находятся в кэше).

Тем не менее, в ходе данного эксперимента не учитывался размер хранимой в кэше информации. Так, для конфигурации сети из 256 слоев по 128 нейронов в каждом, требуется выделить порядка 11 Мб памяти в кэше, при условии сжатия ее посредством `gzip` [**gzip**].

В связи с этим можно сделать вывод, что в кэше следует хранить только большие конфигурации нейронных сетей (более 32 слоев), так как, при меньших размерах конфигурации, кэширование не обеспечивает сопоставимого прироста производительности по времени.

ЗАКЛЮЧЕНИЕ

В результат выполненя данной курсовой работы была достигнута ее цель — спроектирована и разработана база данных для хранения конфигураций нейронных сетей.

В процессе достижения данной цели достижения данной цели были решены следующие задачи:

- проанализированы варианты представления данных и выбран подходящий вариант для решения задачи;
- проанализированы системы управления базами данных и выбрана подходящая система для хранения данных.
- спроектирована базу данных, описаны ее сущности и связи;
- реализован интерфейс для доступа к базе данных;
- реализовано программное обеспечение, позволяющее взаимодействовать со спроектированной базой данных.

В ходе курсовой работы были получены знания в области проектирования баз данных, кжширования данных, проектирования архитектуры web-приложений. Были изучены различные типы баз данных, а так же способы хранения данных, используемые в тех или иных базах данных.

В результате проделанной работы, было разработано программное обеспечение, позволившее снизить время отклика всей системы за счет внедрения механизма кэширования данных.

В ходе выполнения исследовательской части работы было установлено, что кэширование данных всегда приводит к росту производительности системы с точки зрения времени отклика, однако наиболее оптимальным оказался вариант с кэшированием сравнительно больших объемов данных — в этом случае кэширование данных может иметь выигрыш по времени в 39 раз.

ПРИЛОЖЕНИЕ А

Скрипт проведения миграции базы данных

В Листингах А.1 – А.2 приведен скрипт проведения миграции базы данных PostgreSQL версии 14.2 в Docker контейнере.

Листинг А.1 – Скрипт проведения миграции базы данных. Часть 1

```
#!/usr/bin/env bash
set -Eeo pipefail

docker_process_init_files() {
  psql=( docker_process_sql )

  echo
  local f
  for f; do
    export MIGRATION_ID="$(basename $f)"
    case "$f" in
      *.sh)
        if [ -x "$f" ]; then
          echo "$0: running $f"
          "$f"
        else
          echo "$0: sourcing $f"
          . "$f"
        fi
        ;;
      *.sql)    echo "$0: running $f"; docker_process_sql -f "$f";
         echo ;;
      *.sql.gz) echo "$0: running $f"; gunzip -c "$f" |
         docker_process_sql; echo ;;
      *.sql.xz) echo "$0: running $f"; xzcat "$f" |
         docker_process_sql; echo ;;
      *)        echo "$0: ignoring $f" ;;
    esac
    echo
  done
}
```

Листинг А.2 – Скрипт проведения миграции базы данных. Часть 2

```
docker_process_sql() {
  local query_runner=( \
    psql \
    -v ON_ERROR_STOP=1 \
    -v AUTOCOMMIT=off \
    -v MIGRATION_ID=$MIGRATION_ID \
    --no-psqlrc \
    "host=$POSTGRES_HOST \
     dbname=$POSTGRES_DB \
     user=$POSTGRES_USER \
     password=$POSTGRES_PASSWORD" \
  )

  PGHOST= PGHOSTADDR= "${query_runner[@]}" "$@"
}

SCRIPT_DIR=$( cd -- "$( dirname -- "${BASH_SOURCE[0]}" )" &> \
              /dev/null && pwd )/init

echo $SCRIPT_DIR

docker_process_init_files $SCRIPT_DIR/*
```

ПРИЛОЖЕНИЕ Б

Создания таблиц с информацией о пользователях

В Листинге Б.1 приведен скрипт создания таблицы, содержащей информацию о пользователях.

Листинг Б.1 – Создание таблицы с информацией о пользователях

```
--
-- Initialize database with basic tables
--
CREATE TABLE IF NOT EXISTS migrations (
  id VARCHAR PRIMARY KEY
);

SELECT EXISTS (
  SELECT id FROM migrations WHERE id = :'MIGRATION_ID'
) as migrated \gset

\if :migrated
  \echo 'migration' :MIGRATION_ID 'already exists, skipping'
\else
  \echo 'migration' :MIGRATION_ID 'does not exist'

  CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

  CREATE TABLE users_info (
    id              UUID PRIMARY KEY DEFAULT uuid_generate_v4()
    , username      VARCHAR(64)  UNIQUE NOT NULL
    , email         VARCHAR(64)  UNIQUE
    , fullname      VARCHAR(256) NOT NULL
    , password_hash VARCHAR(256) NOT NULL
    , flags         NUMERIC      CHECK (flags > 0)
    , blocked       TIMESTAMP
    , created_at    TIMESTAMP DEFAULT NOW()
    , updated_at    TIMESTAMP DEFAULT NOW()
  );

  INSERT INTO migrations(id) VALUES (:'MIGRATION_ID');
\endif

COMMIT;
```

ПРИЛОЖЕНИЕ В
Создание таблиц с информацией о конфигурации нейронных сетей

В Листингах В.1, В.2 и В.3 приведен скрипт создания таблиц, содержащей информацию о нейронной сети, её структуре и конфигурациях весов.

Листинг В.1 – Создание таблиц с информацией о конфигурации нейронных сетей. Часть 1

```
--
-- Initialize database with models info
--
SELECT EXISTS (
  SELECT id FROM migrations WHERE id = :'MIGRATION_ID'
) as migrated \gset

\if :migrated
  \echo 'migration'   :MIGRATION_ID 'already exists, skipping'
\else
  \echo 'migration' :MIGRATION_ID 'does not exist'

  CREATE TABLE models (
    id           UUID PRIMARY KEY DEFAULT uuid_generate_v4()
    , title      VARCHAR(64) UNIQUE NOT NULL
    , created_at TIMESTAMP   DEFAULT NOW()
    , updated_at TIMESTAMP   DEFAULT NOW()
    , owner_id   UUID        NOT NULL
    , FOREIGN KEY (owner_id)
      REFERENCES users_info(id)
      ON DELETE CASCADE
  );

  CREATE TABLE structures (
    id         UUID        PRIMARY KEY DEFAULT uuid_generate_v4()
    , title    VARCHAR(64) NOT NULL
    , model_id UUID        NOT NULL
    , FOREIGN KEY (model_id)
      REFERENCES models(id)
      ON DELETE CASCADE
  );
```

Листинг B.2 – Сценарий второй миграции базы данных. Часть 2

```
CREATE TABLE layers (
  id               UUID PRIMARY KEY DEFAULT uuid_generate_v4()
  , layer_id       INT
  , limit_func     VARCHAR(64) NOT NULL
  , activation_func VARCHAR(64) NOT NULL
  , structure_id   UUID NOT NULL
  , FOREIGN KEY (structure_id)
    REFERENCES structures(id)
    ON DELETE CASCADE
);

CREATE TABLE neurons (
  id          UUID  PRIMARY KEY DEFAULT uuid_generate_v4()
  , neuron_id INT NOT NULL
  , layer_id  UUID  NOT NULL
  , FOREIGN KEY (layer_id)
    REFERENCES layers(id)
    ON DELETE CASCADE
);

CREATE TABLE neuron_links (
  id        UUID PRIMARY KEY DEFAULT uuid_generate_v4()
  , link_id INT  NOT NULL
  , from_id UUID
  , to_id   UUID NOT NULL
  , FOREIGN KEY (from_id)
    REFERENCES neurons(id)
    ON DELETE CASCADE
  , FOREIGN KEY (to_id)
    REFERENCES neurons(id)
    ON DELETE CASCADE
);

CREATE TABLE weights_info (
  id             UUID PRIMARY KEY DEFAULT uuid_generate_v4()
  , name         VARCHAR(64) NOT NULL
  , created_at   TIMESTAMP   DEFAULT NOW()
  , updated_at   TIMESTAMP   DEFAULT NOW()
  , structure_id UUID        NOT NULL
  , FOREIGN KEY (structure_id)
```

Листинг В.3 – Сценарий второй миграции базы данных. Часть 3

```
    ON DELETE CASCADE
);

CREATE TABLE neuron_offsets (
  id                UUID PRIMARY KEY DEFAULT uuid_generate_v4()
  , value           REAL NOT NULL
  , weights_info_id UUID NOT NULL
  , neuron_id       UUID NOT NULL
  , FOREIGN KEY (weights_info_id)
    REFERENCES weights_info(id)
    ON DELETE CASCADE
  , FOREIGN KEY (neuron_id)
    REFERENCES neurons(id)
    ON DELETE CASCADE
);

CREATE TABLE link_weights (
  id                UUID PRIMARY KEY DEFAULT uuid_generate_v4()
  , value           REAL NOT NULL
  , weights_info_id UUID NOT NULL
  , link_id         UUID NOT NULL
  , FOREIGN KEY (weights_info_id)
    REFERENCES weights_info(id)
    ON DELETE CASCADE
  , FOREIGN KEY (link_id)
    REFERENCES neuron_links(id)
    ON DELETE CASCADE
);

INSERT INTO migrations(id) VALUES (:'MIGRATION_ID');
```

ПРИЛОЖЕНИЕ Г
Создание ролевой модели

В Листингах Г.1, Г.2, Г.3 и Г.4 приведен скрипт конфигурации ролевой модели базы данных, а именно - создания и настройки соответствующих прав доступа для трех ролей:

- пользователь;
- аналитик;
- администратор.

Листинг Г.1 – Сценарий третьей миграции базы данных. Часть 1

```
--
-- Create database roles for statistics, backend admins and users
--
SELECT EXISTS (
  SELECT id FROM migrations WHERE id = :'MIGRATION_ID'
) as migrated \gset

\if :migrated
  \echo 'migration' :MIGRATION_ID 'already exists, skipping'
\else
  \echo 'migration' :MIGRATION_ID 'does not exist'

  \set STAT_USERNAME    `echo $STAT_USERNAME`
  \set STAT_PASSWORD    `echo $STAT_PASSWORD`
  \set ADMIN_USERNAME   `echo $ADMIN_USERNAME`
  \set ADMIN_PASSWORD   `echo $ADMIN_PASSWORD`
  \set REGULAR_USERNAME `echo $REGULAR_USERNAME`
  \set REGULAR_PASSWORD `echo $REGULAR_PASSWORD`

  \set exit_error false

  SELECT (:'STAT_USERNAME' = '') as is_empty \gset
  \if :is_empty
    \warn 'STAT_USERNAME is empty'
    \set exit_error true
  \endif
```

Листинг Г.2 – Сценарий третьей миграции базы данных. Часть 2

```
SELECT (:'STAT_PASSWORD' = '') as is_empty \gset
\if :is_empty
  \warn 'STAT_PASSWORD is empty'
  \set exit_error true
\endif

SELECT (:'ADMIN_USERNAME' = '') as is_empty \gset
\if :is_empty
  \warn 'ADMIN_USERNAME is empty'
  \set exit_error true
\endif

SELECT (:'ADMIN_PASSWORD' = '') as is_empty \gset
\if :is_empty
  \warn 'ADMIN_PASSWORD is empty'
  \set exit_error true
\endif

SELECT (:'REGULAR_USERNAME' = '') as is_empty \gset
\if :is_empty
  \warn 'REGULAR_USERNAME is empty'
  \set exit_error true
\endif

SELECT (:'REGULAR_PASSWORD' = '') as is_empty \gset
\if :is_empty
  \warn 'REGULAR_PASSWORD is empty'
  \set exit_error true
\endif

\if :exit_error
  DO $$
  BEGIN
  RAISE EXCEPTION 'all required environment variables must not be
      empty';
  END;
  $$;
\endif
```

Листинг Г.3 – Сценарий третьей миграции базы данных. Часть 3

```
CREATE ROLE users;

CREATE USER :STAT_USERNAME
  WITH ENCRYPTED PASSWORD :'STAT_PASSWORD'
  IN ROLE users;

GRANT SELECT (id, created_at, updated_at)
  ON TABLE users_info, models, weights_info
  TO :STAT_USERNAME;

CREATE USER :REGULAR_USERNAME
  WITH ENCRYPTED PASSWORD :'REGULAR_PASSWORD'
  IN ROLE users;

GRANT SELECT, INSERT (
    id
    , username
    , email
    , fullname
    , password_hash
    , flags
    , blocked
  )
  ON TABLE users_info
  TO :REGULAR_USERNAME;

GRANT SELECT, INSERT, UPDATE (id, title, owner_id)
  ON TABLE models
  TO :REGULAR_USERNAME;

GRANT DELETE
  ON TABLE models
  TO :REGULAR_USERNAME;
```

Листинг Г.4 – Сценарий третьей миграции базы данных. Часть 4

```
  GRANT SELECT, INSERT, UPDATE, DELETE
    ON TABLE
      structures,
      layers,
      neurons,
      neuron_links,
      weights_info,
      neuron_offsets,
      link_weights
    TO :REGULAR_USERNAME;

  CREATE ROLE administrators;
  GRANT USAGE ON SCHEMA public TO administrators;
  GRANT SELECT, INSERT, UPDATE, DELETE
    ON ALL TABLES
    IN SCHEMA public
    TO administrators;

  ALTER DEFAULT PRIVILEGES IN SCHEMA public
    GRANT SELECT, INSERT, UPDATE, DELETE
    ON TABLES TO administrators;

  CREATE USER :ADMIN_USERNAME
    WITH
    CREATEDB
    CREATEROLE
    ENCRYPTED PASSWORD :'ADMIN_PASSWORD'
    IN ROLE administrators;

  INSERT INTO migrations(id) VALUES (:'MIGRATION_ID');
\endif

COMMIT;
```

ПРИЛОЖЕНИЕ Д
Создание триггеров базы данных

В Листинге Д.1 приведен скрипт создания индексов базы данных.

Листинг Д.1 – Сценарий четвертой миграции базы данных. Часть 1

```
--
-- Initialize database with basic triggers
--
CREATE TABLE IF NOT EXISTS migrations (
    id VARCHAR PRIMARY KEY
);

SELECT EXISTS (
  SELECT id FROM migrations WHERE id = :'MIGRATION_ID'
) as migrated \gset

\if :migrated
  \echo 'migration' :MIGRATION_ID 'already exists, skipping'
\else
  \echo 'migration' :MIGRATION_ID 'does not exist'

  CREATE FUNCTION user_info_preupdate()
  RETURNS trigger AS
  $$
  BEGIN
      NEW.UPDATED_AT = NOW();
      RETURN NEW;
  END;
  $$
  LANGUAGE 'plpgsql';

  CREATE TRIGGER updt_user_info BEFORE UPDATE
  ON users_info
  FOR ROW
  EXECUTE PROCEDURE user_info_preupdate();
```

Листинг Д.2 – Сценарий четвертой миграции базы данных. Часть 2

```
CREATE FUNCTION model_preupdate()
RETURNS trigger AS
$$
BEGIN
    NEW.UPDATED_AT = NOW();
    RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';

CREATE TRIGGER updt_model BEFORE UPDATE
ON models
FOR ROW
EXECUTE PROCEDURE model_preupdate();

CREATE FUNCTION weights_info_preupdate()
RETURNS trigger AS
$$
BEGIN
    NEW.UPDATED_AT = NOW();
    RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';

CREATE TRIGGER updt_weights_info BEFORE UPDATE
ON weights_info
FOR ROW
EXECUTE PROCEDURE weights_info_preupdate();
```

Листинг Д.3 – Сценарий четвертой миграции базы данных. Часть 3

```
CREATE EXTENSION PLPYTHON3U;

CREATE FUNCTION remove_model_from_cache_py()
RETURNS TRIGGER
AS $$
    import os
    import tarantool

    host = os.getenv('CACHE_DB_HOST')
    if host is None:
      return

    port = os.getenv('CACHE_DB_PORT')
    if host is None:
      return

    user = os.getenv('CACHE_DB_USERNAME')
    if user is None:
      return

    pwd = os.getenv('CACHE_DB_PASSWORD')
    if pwd is None:
      return

    model_space = os.getenv('CACHE_DB_MODEL_SPACE')
    if pwd is None:
      return

    try:
      conn = tarantool.connect(host, port,
                               user=user, password=pwd)
      conn.space(model_space).delete(TD["old"]["id"])
    except:
      pass
$$ LANGUAGE PLPYTHON3U;
```

Листинг Д.4 – Сценарий четвертой миграции базы данных. Часть 4

```
CREATE FUNCTION remove_weight_from_cache_py()
RETURNS TRIGGER
AS $$
    import os
    import tarantool

    host = os.getenv('CACHE_DB_HOST')
    if host is None:
      return

    port = os.getenv('CACHE_DB_PORT')
    if host is None:
      return

    user = os.getenv('CACHE_DB_USERNAME')
    if user is None:
      return

    pwd = os.getenv('CACHE_DB_PASSWORD')
    if pwd is None:
      return

    weight_space = os.getenv('CACHE_DB_WEIGHT_SPACE')
    if pwd is None:
      return

    try:
      conn = tarantool.connect(host, port,
                               user=user, password=pwd)
      conn.space(weight_space).delete(TD["old"]["id"])
    except:
      pass
$$ LANGUAGE PLPYTHON3U;
```

Листинг Д.5 – Сценарий четвертой миграции базы данных. Часть 5

```
CREATE TRIGGER rm_model_upd AFTER UPDATE
ON models FOR ROW
EXECUTE PROCEDURE remove_model_from_cache_py();

CREATE TRIGGER rm_model_del AFTER DELETE
ON models FOR ROW
EXECUTE PROCEDURE remove_model_from_cache_py();

CREATE TRIGGER rm_model_s_upd AFTER UPDATE
ON structures FOR ROW
EXECUTE PROCEDURE remove_model_from_cache_py();

CREATE TRIGGER rm_model_s_del AFTER DELETE
ON structures FOR ROW
EXECUTE PROCEDURE remove_model_from_cache_py();

CREATE TRIGGER rm_model_wi_upd AFTER UPDATE
ON weights_info FOR ROW
EXECUTE PROCEDURE remove_model_from_cache_py();

CREATE TRIGGER rm_model_wi_del AFTER DELETE
ON weights_info FOR ROW
EXECUTE PROCEDURE remove_model_from_cache_py();

CREATE TRIGGER rm_weight_del AFTER DELETE
ON models FOR ROW
EXECUTE PROCEDURE remove_weight_from_cache_py();

CREATE TRIGGER rm_weight_s_upd AFTER UPDATE
ON structures FOR ROW
EXECUTE PROCEDURE remove_weight_from_cache_py();

CREATE TRIGGER rm_weight_s_del AFTER DELETE
ON structures FOR ROW
EXECUTE PROCEDURE remove_weight_from_cache_py();

CREATE TRIGGER rm_weight_wi_upd AFTER UPDATE
ON weights_info FOR ROW
EXECUTE PROCEDURE remove_weight_from_cache_py();
```

Листинг Д.6 – Сценарий четвертой миграции базы данных. Часть 6

```
  CREATE TRIGGER rm_weight_wi_del AFTER DELETE
  ON weights_info FOR ROW
  EXECUTE PROCEDURE remove_weight_from_cache_py();

  INSERT INTO migrations(id) VALUES (:'MIGRATION_ID');
\endif

COMMIT;
```

ПРИЛОЖЕНИЕ Е
Создания индексов базы данных

В Листинге Е.1 приведен скрипт создания индексов базы данных.

Листинг Е.1 – Сценарий пятой миграции базы данных

```
--
-- Initialize database with basic tables
--
CREATE TABLE IF NOT EXISTS migrations (
  id VARCHAR PRIMARY KEY
);

SELECT EXISTS (
  SELECT id FROM migrations WHERE id = :'MIGRATION_ID'
) as migrated \gset

\if :migrated
  \echo 'migration' :MIGRATION_ID 'already exists, skipping'
\else
  \echo 'migration' :MIGRATION_ID 'does not exist'

  CREATE UNIQUE INDEX idx_struct_model ON structures(model_id);

  CREATE INDEX idx_layer_struct ON layers(structure_id);

  CREATE INDEX idx_neurons_layer ON neurons(layer_id);

  CREATE INDEX idx_link_from_neurons ON neuron_links(from_id);

  CREATE INDEX idx_link_to_neurons ON neuron_links(to_id);

  CREATE INDEX idx_weights_struct ON weights_info(structure_id);

  CREATE INDEX idx_off_weights ON neuron_offsets(weights_info_id);

  CREATE INDEX idx_lw_weights ON link_weights(weights_info_id);
\endif

COMMIT;
```

ПРИЛОЖЕНИЕ Ж
Развертывание приложения

В Листинге Ж.1 приведен скрипт создания `docker`-образа базы данных.

Листинг Ж.1 – Dockerfile для базы данных

```
FROM postgres:14.2
LABEL maintainer="Gregory @migregal Mironov"

RUN apt-get update \
    && apt-get -y install \
        python3 \
        python3-pip \
        postgresql-plpython3-14 \
    && pip3 install tarantool\>0.4
```

Для развертывания использовался `docker-compose`. Данная система позволяет реализовать наследование конфигураций, что позволяет, например, реализовать множество конфигураций развертывания, указав общие для них элементы лишь единожды. В Листингах Ж.2 – Ж.4 представлена конфигурация общих элементов для конфигураций развертывания приложения.

Листинг Ж.2 – Конфигурация общих элементов развертывания. Часть 1

```
version: '3.9'

services:
  postgres_db:
    build:
      context: ./database
    container_name: postgresql
    env_file: ./database/.env
    restart: always
    healthcheck:
      test:
        - "CMD-SHELL"
        - "pg_isready -d $$POSTGRES_DB -U $$POSTGRES_USER"
      interval: 5s
      timeout: 5s
      retries: 5
```

Листинг Ж.3 – Конфигурация общих элементов развертывания. Часть 2

```
  migration:
    build:
      context: ./migration
    container_name: db-migration
    working_dir: /usr/app
    entrypoint: ./migrate
    env_file: ./migration/.env
    volumes:
      - ../database/migration:/usr/app
      - ../database/migration/init/:/docker-entrypoint-initdb.d/

  tarantool-db:
    image: tarantool/tarantool:2.8
    container_name: tarantool-db
    command: tarantool /cache/init.lua
    env_file:
      - ./cache/.env
    volumes:
      - ./cache:/cache

  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    volumes:
      - ./prometheus/:/etc/prometheus/
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/usr/share/prometheus/
        console_libraries'
      - '--web.console.templates=/usr/share/prometheus/consoles'
    depends_on:
      - cube
    restart: always
```

Листинг Ж.4 – Конфигурация общих элементов развертывания. Часть 3

```
  grafana:
    image: grafana/grafana-oss:latest
    container_name: grafana
    environment:
      - GF_SERVER_DOMAIN=localhost
      - GF_SERVER_ROOT_URL=http://localhost/grafana/
      - GF_SERVER_SERVE_FROM_SUB_PATH=true
    depends_on:
      - prometheus
    restart: always

  cube:
    env_file: ./cube/.env
    expose:
      - 8080
```

Листингах Ж.5 – Ж.9 представлена итоговая конфигурация развертывания приложения.

Листинг Ж.5 – Конфигурация итоговых элементов развертывания. Часть 1

```
version: '3.9'

services:
  nginx:
    image: nginx:latest
    volumes:
      - ../nginx/nginx.conf:/etc/nginx/nginx.conf:ro
    networks:
      - api-gateaway
      - stat-gateaway
      - backend
    ports:
      - 4000:4000
      - 4001:4001
    depends_on:
      - cube
      - grafana
    restart: always
```

Листинг Ж.6 – Конфигурация итоговых элементов развертывания. Часть 2

```
postgres_db:
  extends:
    file: common-services.yml
    service: postgres_db
  env_file: ./database/.env
  networks:
    - database
  volumes:
    - ./data/postgresql:/var/lib/postgresql/data
  healthcheck:
    test:
      - "CMD-SHELL"
      - "pg_isready -d $$POSTGRES_DB -U $$POSTGRES_USER"
    interval: 5s
    timeout: 5s
    retries: 5

migration:
  extends:
    file: common-services.yml
    service: migration
  env_file: ./database/.env
  networks:
    - database
  depends_on:
    postgres_db:
      condition: service_healthy
  restart: on-failure:3

tarantool-db:
  extends:
    file: common-services.yml
    service: tarantool-db
  env_file:
    - ./cache/.env
  networks:
    - cache
```

Листинг Ж.7 – Конфигурация итоговых элементов развертывания. Часть 3

```
  prometheus:
    extends:
      file: common-services.yml
      service: prometheus
    networks:
      - backend
      - metrics
    expose:
      - 9090
    depends_on:
      - cube
    restart: always

  grafana:
    extends:
      file: common-services.yml
      service: grafana
    volumes:
      - ./grafana/datasources:/etc/grafana/provisioning/datasources
        /
      - ./grafana/dashboards:/etc/grafana/provisioning/dashboards/
    networks:
      - stat-gateaway
      - metrics
    ports:
      - 3000
    depends_on:
      - prometheus
    restart: always
```

Листинг Ж.8 – Конфигурация итоговых элементов развертывания. Часть 4

```
  cube:
    extends:
      file: common-services.yml
      service: cube
    build:
      context: ../
      dockerfile: ./docker/cube/Dockerfile
    volumes:
      - ../out/crypto:/tmp/crypto/
      - ../cube/prod.yml:/tmp/config.yml:ro
    networks:
      - backend
      - database
      - cache
    expose:
      - 10001
    depends_on:
      migration:
        condition: service_completed_successfully
    restart: always
    deploy:
      mode: replicated
      replicas: 3

networks:
  api-gateaway:
    driver: bridge

  stat-gateaway:
    driver: bridge
    internal: true

  metrics:
    driver: bridge
    internal: true

  backend:
    driver: bridge
    internal: true
```

Листинг Ж.9 – Конфигурация итоговых элементов развертывания. Часть 5

```
  cache:
    driver: bridge
    internal: true

  database:
    driver: bridge
    internal: true
```