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 НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

НА ТЕМУ:

«Паттерны проектирования высоконагруженных систем»

Студент ИУ7-51Б (Группа) _____ (Подпись, дата) М. Д. Мицевич (И. О. Фамилия)

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

2022 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

На сегодняшний день существует огромное количество проектов, высоконагруженных данными (data-intensive system). Технологии, реализуемые в подобных проектах, важны во многих прикладных областях: от прогнозирующей аналитики до мониторинга окружающей среды, от электронного правительства до умных городов [1].

Целью данной работы является изучение принципов построения высоконагруженные систем.

Для достижения поставленной цели требуется выполнить следующие задачи:

- определить основные термины, связанные с областью высоконагруженных систем;
- выявить требования, которые к таким системам предъявляются;
- провести классификацию паттернов проектирования таких систем;
- определить преимущества и основные проблемы, связанные с такими паттернами.

1 Анализ предметной области

В данной работе рассматриваются подходы к построению высоко нагруженных данными систем. Это системы, которые могут принимать, обрабатывать и генерировать большие объемы данных разной природы и из разных источников с течением времени, организованные с помощью различных технологий и с целью извлечения ценности из данных для различных типов предприятий [2].

С такими системами связан термин хайлоад. Это нагрузка, с которой не справляется аппаратное обеспечение из-за различных технических ограничений, к примеру, из-за нехватки процессорного времени или памяти [3].

Высоко нагруженные системы должны обладать рядом свойств. Из них наибольшее значение в большинстве программных систем имеют следующие [4]:

- надежность;
- масштабируемость;
- удобство сопровождения.

Надежность определяет устойчивость работы системы как при благоприятных условиях, так и при сбоях. В данном случае сбоем называется отклонение от штатной работы одного или нескольких компонентов системы, но всей целиком [4].

Под масштабируемостью понимают способность системы, процесса или сети расти и справляться с возросшим спросом. Существует два типа масштабирования [2]:

- вертикальное, при таком подходе серверную часть приложения просто переводят на более мощную машину;
- горизонтальное, в этом случаем происходит перераспределение нагрузки с одной машины на несколько.

Сопровождением системы называется исправление ошибок, поддержание работоспособности его подсистем, расследование отказов, адаптацию к новым платформам, модификацию под новые сценарии использования и добавление

новых возможностей [2]. Отсюда следует, что система должна быть способна к добавлению нового функционала или изменению сценариев поведения без внесения изменений в существующий программный код.

Не существует единственного правильного решения для построения высоко нагруженных данными систем, так как каждый подход, выбираемый при проектировании системы, должен быть связан с входными данными и конкретными задачами, решаемыми программным обеспечением. Существуют различные паттерны проектирования таких систем, они будут рассмотрены в следующей главе.

2 Классификация существующих решений

Классификация паттернов, которые будут рассмотрены в этой главе представлена на рисунке 2.1

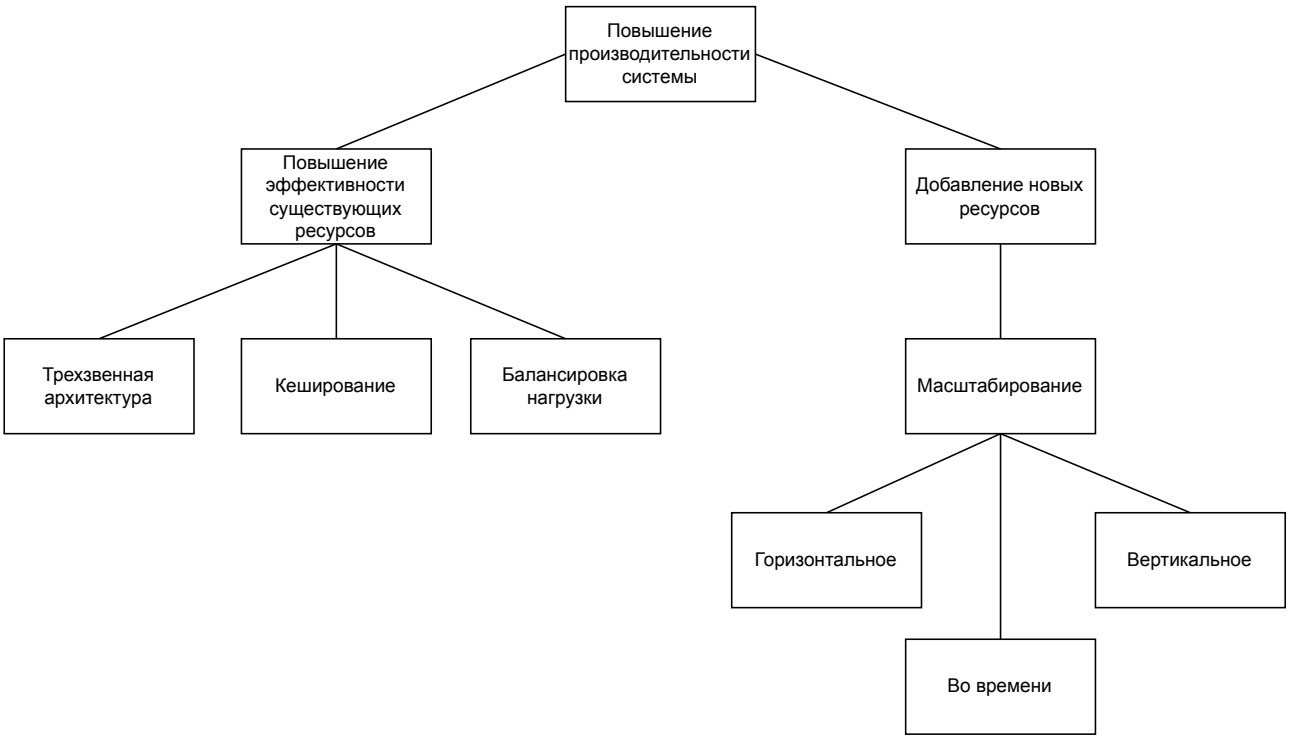

Рисунок 2.1 – Классификация паттернов проектирования высоконагруженных систем

Все паттерны можно поделить на те, которые требуют введение в систему новых ресурсов, и те, которые улучшают производительность системы на основе уже существующих ресурсов. К первой категории относится масштабирование, которое может быть вертикальным, горизонтальным или во времени. Среди паттернов второй категории можно выделить трехзвенную архитектуру, кеширование и балансировку нагрузки.

2.1 Трехзвенная архитектура

На рисунке 2.2 представлена классическая реализация клиент-серверной архитектуры.

Клиент посылает запросы на сервер, он выполняет некоторые вычисления, делает запросы к базе данных и отправляет ответ клиенту. Сервер, как правило, представляет собой тяжеловесную многопоточную программу, нацеленную на массивные вычисления.

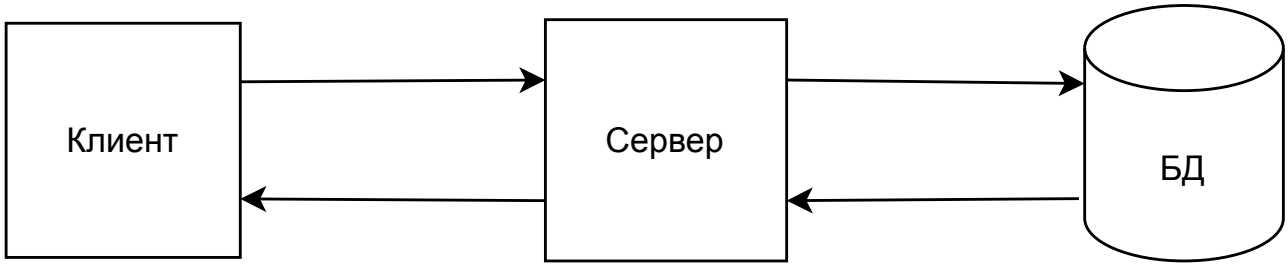

Рисунок 2.2 – Классичическая реализация клиент-серверной архитектуры

Среди запросов, которые поступают на сервер, далеко не все требуют большого числа вычислений для получения ответа. Некоторые из них просто запрашивают статические данные, которые хранятся на диске. Нагружать мощный сервер, который должен производить вычисления, такими запросами – нерационально, поэтому вводится дополнительное звено на пути от клиента к серверу, которое называется фронтендом. Его задачей является фильтрация и обработка такого рода запросов.

Схема трехзвенной архитектуры приведена на рисунке 2.3.

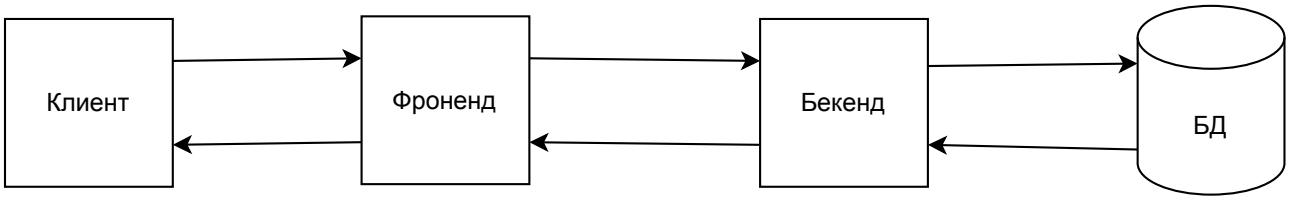

Рисунок 2.3 – Схема трехзвенной архитектуры

Ключевым моментом во фронтенде является то, какое количество ресурсов тратится на обработку одного запроса, поэтому для этого звена используются легковесные сервера [2].

Еще одной функцией фронтенда является буфферизация при обслуживании медленных клиентов, то есть фроненд не будет открывать соединение с бекендом до тех пор, пока не получит весь запрос целиком. Если этого не делать, то бекенд вместо выполнения вычислений будет ожидать получения данных от клиента.

2.2 Балансировка нагрузки

Третьей функцией фронтенда является распределение запросов между бекендами, когда их больше одного. Схема такой архитектуры представлена на рисунке 2.4.

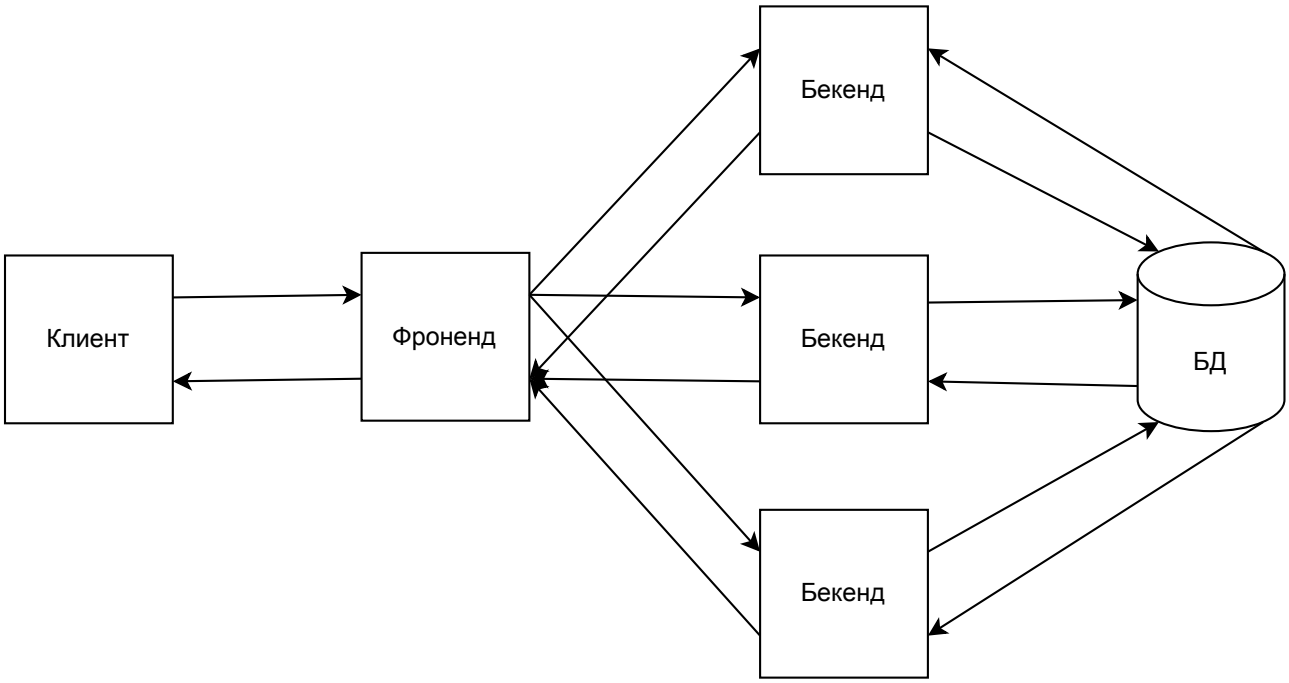

Рисунок 2.4 – Балансировка нагрузки через фронтенд

Стратегия балансировки нагрузки преследует следующие цели [5]:

- оптимальная общая производительность системы, то есть не должно возникать таких ситуация, при которых часть системы работает на пределе своих мощностей, а другая не работает вовсе;
- справедливость обслуживания – все задания должны обслуживаться с одинаковым уровнем привилегий вне зависимости от их происхождения;
- устойчивость к отказам – поддержание производительности системы при наличии частичных отказов.

Существуют различные методы балансировки нагрузки, которые помогают достичь эти цели, их классификация представлена на рисунке 2.5.

2.2.1 Статические методы

Во всех статических методах балансировки вся информация о процессах собирается до их обработки. После назначения процессу обработчика он уже никогда не может перейти к другому [5].

Статические методы можно поделить на детерминированные и вероятностные. Первые назначают задачи по какому-то фиксированному критерию. К примеру, можно высчитывать хеш функцию от IP адреса клиента.

Рисунок 2.5 – Классификация методов балансировка нагрузки

Вероятностные методы распределяют задачи по обработчикам с некоторой вероятностью. Примером такого метода является weighted round robin. При таком подходе каждый сервер имеет заранее определенный вес, который, к примеру, может зависеть от мощностей сервера. Балансировщик нагрузки обходит все серверы по кругу и выдает им задачи в зависимости от их веса.

Главным недостатком статических методов является то, что они при принятии решений никак не учитывают колебания нагрузки на систему [5].

2.2.2 Динамические методы

Стратегия заявок

Загруженный процессор посылает запрос на получение заявок на обработку его задач от других процессоров. Главный процессор получает информацию о заявках и статусе выполнения каждого сервера системы и выбирает, кому передать группу задач, с которой не справляется загруженный узел.

Основной проблемой такого подхода является ситуация, при которой процессор, который принял на себя группу задач, сам перестал справляться с нагрузкой [5]. Для решения этой проблемы некоторые вариации метода заявок позволяют процессору, отправившему заявку на получение группы задач отклонять некоторые задачи и посылать их обратно главному. Для этого должна быть реализована возможность передачи сообщений от процессоров, выполняющих задачи, к основному.

Стратегия отбора

В стратегии заявок часть процессов переходит с одного сервера на другой и возвращается обратно из-за непредсказуемого роста нагрузки на процессор, отправивший заявку. Для решения этой проблемы в стратегии отбора процессы переходят не группами, а по одному.

В стратегии отбора миграцию инициируют малонагруженные процессоры. Каждый из процессоров определяет одно из трех состояний своей загруженности: HLOAD - высоконагруженный, NLOAD - обычная нагрузка или LLOAD - слабая нагрузка [5]. Каждый из серверов хранит таблицу нагрузки других, и, когда его состояние изменяется, он посылает всем остальным оповещение с новым состоянием.

Если сервер переходит в состояние LLOAD, то он выбирает из таблицы загруженный сервер и инициирует миграцию процесса.

Стратегия порога

При таком подходе для каждого сервера выставляется пороговое значение, которое сравнивается с длинной очереди на выполнение при поступлении новой задачи. Длина этой очереди считается как сумма количества задача, которые в данный момент выполняются и тех, что находятся в ожидании получения ресурсов.

Если при поступлении задачи длина очереди равна некотором заранее установленному значению, то случайным образом выбирается другой сервер, которому она будет отправлена. Если у выбранного сервера также достигнуто пороговое значение, то он аналогичным образом передаст задачу другому. Процесс будет продолжаться до тех пор, пока не будет найден подходящий сервер.

Главным преимуществом такого подхода является то, что серверам никак не нужно между собой коммуницировать, что ускоряет процесс обработки [5].

Жадная стратегия

При таком подходе состояние каждого сервера описывается некоторой функцией $f(n)$, где n – число задач на текущем сервере. Если на момент поступления новой задачи число n больше нуля, то начинаются поиски нового

сервера, состояние $f(n)$ которого меньше или равно состоянию текущего. Если такой был найден, то задача передается ему.

Исследования [6] показывают, что жадная стратегия превосходит стратегию порога из-за того, что при жадном подходе происходит попытка передать все задачи, которые приходят на занятый сервер, в то время как при стратегии порога задачи передаются только при достижении какого-то критического значения длины очереди.

2.3 Кеширование

Кеширование - это техника, направленная на повышение производительности системы, за счет временного копирования часто используемых данных в быстрое хранилище, расположенное рядом с приложением [7].

Существует несколько разных подходов к организации кеширования:

- кеш хранится локально на каждой машине, где запущено приложение;
- кеш хранится отдельно на удаленной машине, он один общий для всех машин.

Одной из основных проблем глобального кеша является его обновление. Общая схема кеширования представлена на рисунке 2.6.

Рисунок 2.6 – Общая схема организации кеширования

Проблемы в такой системе возникают, когда сразу несколько процессов запрашивают данные, которых нет в кеше, и начинают их вычислять параллельно. В таких случаях начинает расти нагрузка на базу данных, так как к ней приходят запросы на одни и те же данные, что повышает нагрузку на систему целиком.

Для предотвращения этой проблемы процесс, который начал вычисления, должен выставить некий флаг, сообщающий другим процессам, что данные вычисляются и через какое-то время появятся в кеше.

Также требуется ввести ограничение на время вычисления. Если процесс, установивший флаг, не вычислил данные в кеше за отведенное время, значит в что-то сработало не в штатном режиме, к примеру, процесс мог аварийно завершиться. В таком случае первый процесс, который обнаружит эту задержку должен взять вычисления на себя и обновить соответствующий флаг.

Схема такой обработки кеша приведена на рисунке 2.7.

Еще одной проблемой кеширования данных является зависимость разных блоков от одной и той же единицы информации (кешировать можно не

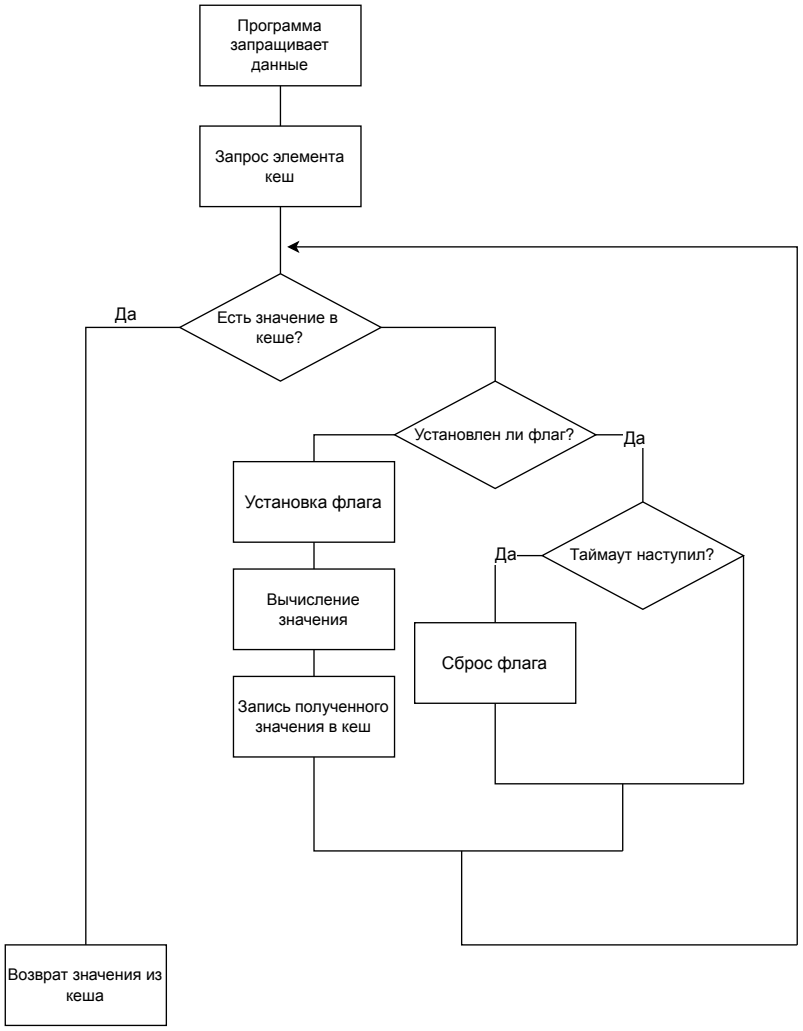

Рисунок 2.7 – Кеширование с флагом о вычислении

только отдельные данные, но и блоки целиком). Возникает вопрос, как при обновлении единицы информации обновить все блоки, которые от нее зависят?

Для решения этой проблемы существуют два различных способа: сброс кешей при наступлении определенного события и тегирование кешей.

При первом подходе составляется граф, который определяет какие блоки зависят от каждой единицы данных. При обновлении информации каждого куска сбрасываются все блоки, которые от него зависят.

Второй вариант основан на создании обратного графа, который определяет от каких кусков данных зависит каждый блок. Каждая единица информации подвергается тегированию, то есть помимо полезных данных она должна хранить текущую версию или время создания. В таком случаем при запросе каждого блока информации идет сравнение версий или времени обновления каждого из его кусков с их оригинальными версиями, если оно меньше, то этот кусок обновляется в блоке.

Также важно заполнять кеш, к которому идет наибольшее число обращений, перед началом стартом работы сервера. Нужно это для предотвращения проблемы запуска с "непрогретмым"кешом, когда при перезапуске сервера все запросы начинают высчитывать данные в кеше.

2.4 Масштабирование

2.4.1 Вертикальное масштабирование

Принцип вертикального масштабирования заключается в замене имеющихся ресурсов на более мощные. По проведенным исследованиям [8] стоимость не масштабируется линейно относительно введенных ресурсов. Также такой подход сталкивается с ограничением мощностей вычислительных машин.

2.4.2 Горизонтальное масштабирование

Суть горизонтального масштабирования заключается не в замене имеющихся машин на более мощные, а в добавлении новых и распределении нагрузки между ними. Подходы к балансировке нагрузки были рассмотрены в разделе 2.2.

При горизонтальном масштабировании не всегда будет наблюдаться линейный рост производительности от количества добавленных в систему ресурсов, так как узлы системы должны обмениваться между собой сообщениями для получения общего результата работы [8]. Такое поведение подчиняется закону Амдала, согласно которому ускорение выполнения программы за счёт распараллеливания её инструкций на множестве вычислителей ограничено временем, необходимым для выполнения её последовательных инструкций.

Горизонтальное масштабирование возможно только для stateless систем - таких, которые не изменяют и не сохраняют свое состояние после выполнения запроса [9].

Также возможно запускать на разных серверах не копии приложения, а его отдельные функциональные части.

Горизонтальное масштабирование может применяться к любому из звеньев трехзвенной структуры.

2.4.3 Масштабирование во времени

Далеко не все запросы надо выполнять сразу после их поступления. При поступлении некоторых рода задач можно ставить их в некую очередь, которая будет постепенно просматриваться обработчиком. При таком подходе клиент получает ответ, что его запрос обрабатывается, и соединение с ним закрывается.

Также очереди можно использовать для межсервисной коммуникации. Некий сервис А ставит задачу сервису Б и добавляет ее в свою очередь задач. Специальная программа разбирает эту очередь и пытается отправить сообщение сервису Б и ждет от него ответа. Задача убирается из очереди только тогда, когда от сервиса Б пришел ответ, что сообщение обработано, иначе попытка повторяется.

2.5 Вывод

Каждый из проанализированных используется в зависимости от поставленных целей и входных данных.

Трехзвенную архитектуру стоит применять, когда существует возможность обслуживания медленных клиентов, есть запросы, которые не требуют вычислений, к примеру, отдача статики и когда предполагается масштабирование бекендов.

Кеширование повышает производительность системы, когда есть данные, которые не часто изменяются, но может наоборот замедлить работу системы, когда данные изменяются часто. Также если вычисления стоят дешевле, чем хранение информации, то кеширование нецелесообразно.

ЗАКЛЮЧЕНИЕ

На основе проведенного исследования можно сделать вывод о том, что не существует единственного верного решения проектирования высоконагруженных систем. Существуют различные паттерны, использования которых зависит от конкретных задач, поставленных системе, и обрабатываемых ею данных.

В рамках работы были выполнены все поставленные задачи. Цель работы была достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Casale G.* Dice: quality- driven development of data-intensive cloud applications. — Proceedings of the Seventh International Workshop on Modeling in Software Engineering. IEEE Press, 2015.
2. *Амиров С.* Особенности разработки высоконагруженных систем. — International Journal of Open Information Technologies, 2020. — с. 9.
3. Блог Highload Junior. — Дата обращения: 10.12.2020. Режим доступа: `https://highload.guide/blog/highload-for-beginners.html`.
4. *Kleppmann M.* Designing Data-Intensive Applications. — Edition. Sebastopol: O'Reilly Media, 2017. — с. 640.
5. *Alakeel A.* Load Balancing in Distributed Computer Systems. — International Journal of Computer Science, Information Security, 2010. — с. 7.
6. *Chowdhury S.* The Greedy Load Sharing Algorithms. — Distributed Comput., 1990.
7. Caching in distributed systems. — Дата обращения: 10.12.2020. Режим доступа: `https://docs.microsoft.com/en-us/azure/architecture/best-practices/caching`.
8. *Henderson S.* Building Scalable Web Sites. — Edition, Sebastopol, O'Reilly Media, Inc, 2006.
9. *Burns B.* Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services. — O'Reilly Media, Inc., 2018. — с. 164.

ПРИЛОЖЕНИЕ А

Презентация

Слайды презентации представлены на рисунках А.1 - А.2

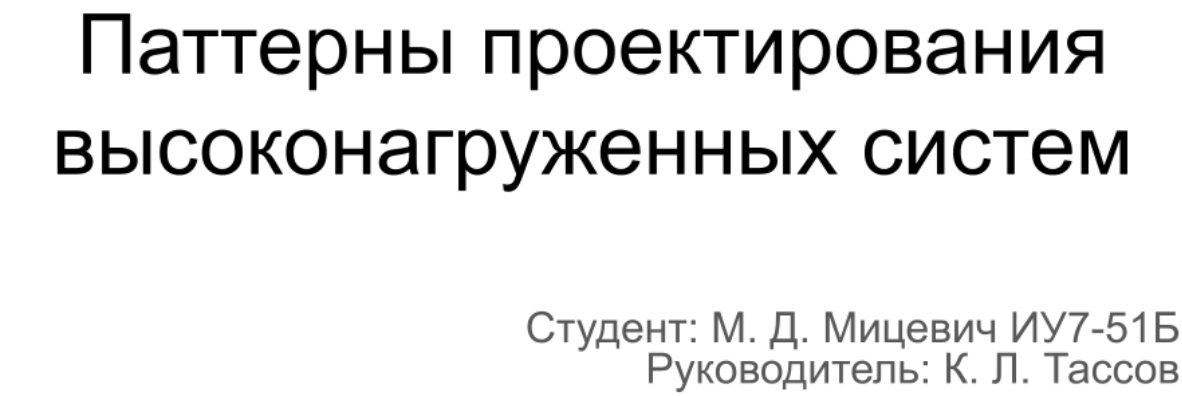

Рисунок А.1 – Слайд 1

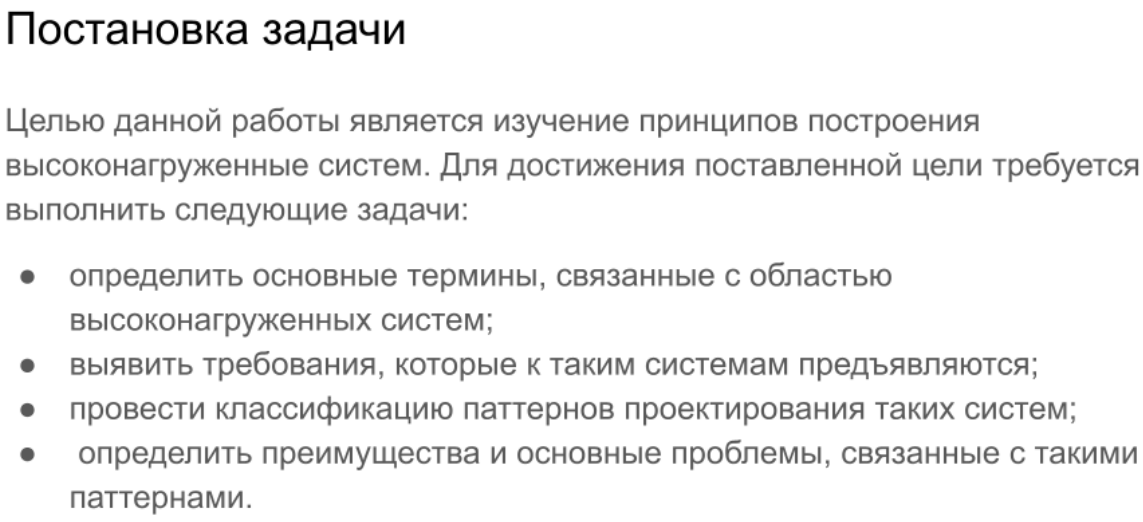

Рисунок А.2 – Слайд 2

Классификация паттернов

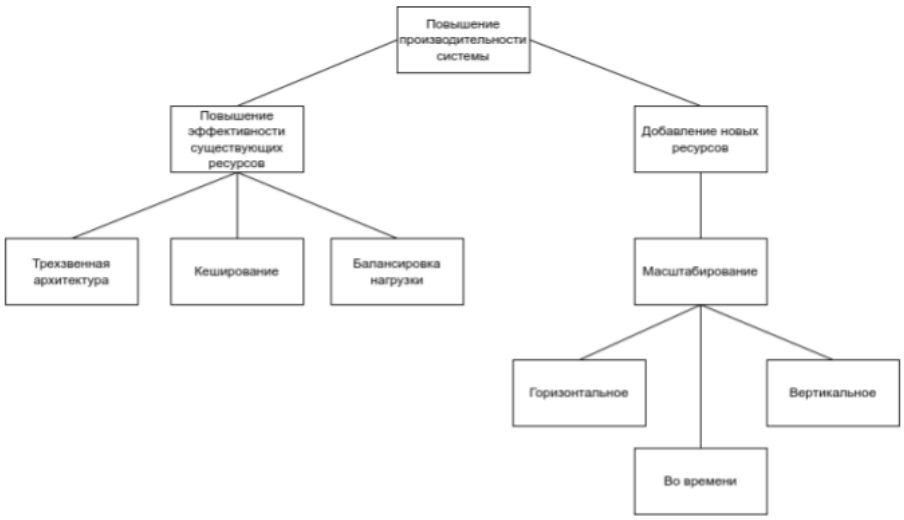

Рисунок А.3 – Слайд 3

Трехзвенная архитектура

- Обработка простых запросов
- Проверка валидности запросов
- Обслуживание медленных клиентов
- масштабирование бекендов

Рисунок А.4 – Слайд 4

Балансировка нагрузки

Рисунок А.5 – Слайд 5

Кеширование

- это техника, направленная на повышение производительности системы, за счет временного копирования часто используемых данных в быстрое хранилище, расположенное рядом с приложением

Рисунок А.6 – Слайд 6

Кеширование (2)

Рисунок А.7 – Слайд 7

Масштабирование

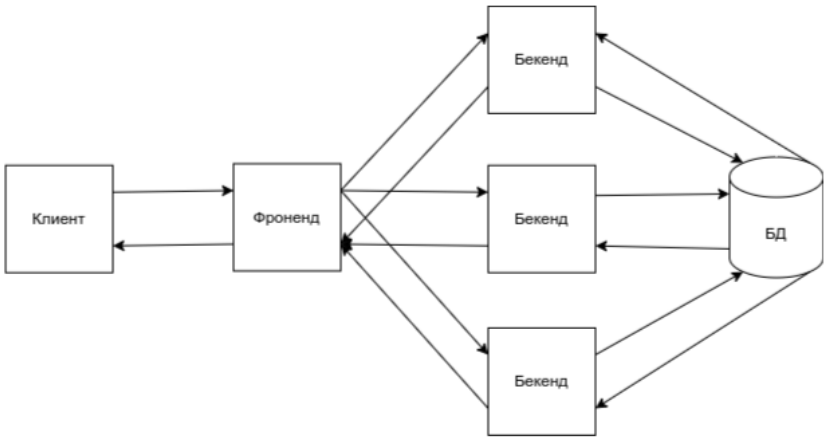

Рисунок А.8 – Слайд 8