МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
БУРЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ ДОРЖИ БАНЗАРОВА

И.С.Поломошнов, О.А.Литвиненко

«Оптимизация веб-приложений»

Рекомендовано Экспертным советом университета в качестве учебного пособия для обучающихся по направлениям подготовки
09.03.03 «Прикладная информатика»,
10.05.03 «Безопасность открытых информационных систем»,
02.03.03 Математическое обеспечение и администрирование информационных систем
09.02.01.02 «Компьютерные системы и комплексы»,
09.02.07.02 «Информационные системы и программирование»

Улан-Удэ
Издательство Бурятского госуниверситета им. Доржи Банзарова
2026

УДК _____
ББК _____

Утверждено к печати Экспертным советом университета
Протокол № __ от «___» _____ 2026 г.

Рецензенты
А.В.Бадеев
Кандидат физико-математических наук, доцент кафедры информационной безопасности Института математики, физики и компьютерных наук БГУ
А.В.Лобанов
Доцент практики, начальник отдела информационной безопасности ГБУ РБ «Информационно-Технологический Центр»

И.С.Поломошнов, О.А.Литвиненко
Оптимизация веб-приложений: учебное пособие.
— Улан-Удэ: Издательство Бурятского госуниверситета имени Доржи Банзарова, 2026. — 155 с.
ISBN

В учебном пособии системно рассмотрены фундаментальные основы и инженерные практики повышения производительности веб-приложений. Детально излагаются ключевые метрики пользовательского опыта (Core Web Vitals), математические модели сетевых и вычислительных ограничений, а также стратегии оптимизации клиентской стороны (управление критическим путём рендеринга, декомпозиция кода, кэширование) и серверной инфраструктуры (балансировка нагрузки, оптимизация баз данных, эволюция сетевых протоколов). Особое внимание уделяется методологии «доказательной оптимизации», интеграции контроля производительности в процессы CI/CD и интерпретации результатов нагрузочного тестирования. Материал направлен на формирование у обучающихся навыков измерения, анализа и улучшения показателей производительности распределённых систем с использованием строгих инженерных методов.

Пособие предназначено для обучающихся по направлению подготовки 09.03.03 Прикладная информатика, 01.03.02 Прикладная математика и информатика, 09.03.02 Информационные системы и технологии, 02.03.03 Математическое обеспечение и администрирование информационных систем, 10.05.03 Безопасность открытых информационных систем, 09.02.01.02 Компьютерные системы и комплексы, 09.02.07.02 Информационные системы и программирование, 09.02.11.02 Разработка и управление программным обеспечением

УДК _____
ББК _____

© И.С.Поломошнов, 2026
© О.А.Литвиненко, 2026
© Бурятский госуниверситет
имени Доржи Банзарова, 2026

ПРЕДИСЛОВИЕ

Настоящее учебное пособие «Оптимизация веб-приложений» представляет собой систематизированное изложение теоретических основ и практических методов повышения производительности распределённых информационных систем. Пособие предназначено для студентов высших учебных заведений, обучающихся по направлениям подготовки 09.03.03 «Прикладная информатика», 02.03.03 «Математическое обеспечение и администрирование информационных систем», 09.03.04 «Программная инженерия», а также по смежным техническим и математическим специальностям. Материал будет полезен преподавателям, ведущим соответствующие дисциплины, практикующим разработчикам и всем специалистам, стремящимся к углублённому пониманию нефункциональных аспектов разработки программного обеспечения.

Целью освоения дисциплины является формирование у обучающихся целостной системы знаний о метриках, моделях и инженерных практиках, обеспечивающих высокую производительность и отказоустойчивость веб-приложений в условиях неопределённости сетевой среды и высокой конкурентной нагрузки.

В результате изучения материалов пособия студент должен:

Знать:

- классификацию и семантику ключевых метрик производительности (Core Web Vitals, пользовательские и бизнес-ориентированные KPI);
- физические и математические ограничения, лежащие в основе сетевого взаимодействия, включая законы распространения сигнала и модели теории массового обслуживания;
- архитектурные паттерны и технологические решения для оптимизации клиентской (Critical Rendering Path, Tree Shaking, виртуализация DOM) и серверной (нормализация/денормализация БД, балансировка нагрузки, бессерверные вычисления) частей приложения.

Уметь:

- осуществлять комплексную диагностику производительности веб-приложений посредством синтеза лабораторных измерительных методик (инструментальные панели Lighthouse, платформа WebPageTest) и систем пассивного мониторинга реального пользовательского опыта (Real User Monitoring, RUM), выполняя квалифицированную интерпре-

тацию получаемых количественных данных с выявлением статистически значимых аномалий и трендов;

- задействовать формальный аппарат теории массового обслуживания и вычислительной математики для априорного прогнозирования временны́х характеристик отклика распределённой системы, идентификации потенциальных «бутылочных горлышек» на базе марковской модели M/M/1, а также для оценки предельно достижимого ускорения вычислений с применением закона Амдала;
- проектировать и имплементировать многоуровневые стратегии кэширования, охватывающие клиентский слой (браузерный кэш, Service Worker), граничные узлы сетей доставки контента (CDN) и серверный уровень (In-Memory Data Grids, кэш приложений), аргументированно специфицируя время жизни кэшируемых объектов (TTL) и механизмы принудительной инвалидации на основе событийной модели или пороговых значений.

Владеть:

- практическими компетенциями в области профилирования как клиентского программного кода (JavaScript, библиотека React с использованием React DevTools Profiler и Chrome Performance Insights), так и серверной среды исполнения (платформа Node.js с применением инструментария Clinic.js и встроенного сэмплирующего профайлера) для прецизионной локализации вычислительно-избыточных участков, ответственных за деградацию показателей отзывчивости;
- техниками автоматизации рутинных операций по обеспечению производительности в рамках пайплайнов непрерывной интеграции и доставки (CI/CD), включая конвейерное сжатие статических ресурсов (минификация, транспортное сжатие Brotli), автоматизированный статический анализ состава и размера генерируемых бандлов (bundle auditing) и контроль соблюдения утверждённых бюджетов производительности (Performance Budgets) как блокирующего критерия при слиянии изменений;
- методологией организации и проведения нагрузочного тестирования серверных компонентов с использованием профильных инструментов генерации трафика (Grafana k6, Artillery.io), включая разработку параметризованных сценариев, моделирующих вариативные паттерны пользовательского поведения, и последующую аналитическую обработку агрегированных результатов (расчёт перцентилей задержки, по-

строение кривых пропускной способности) для выработки обоснованных рекомендаций по эластичному масштабированию инфраструктурных ресурсов.

ВВЕДЕНИЕ

Современная цифровая экосистема функционирует в режиме перманентного экспоненциирования объёмов циркулирующих данных: согласно отраслевым замерам, глобальный интернет-трафик преодолел рубеж в четыре зеттабайта в 2023 году. Данное обстоятельство формирует качественно иные, беспрецедентные по своей строгости императивы к временны́м и ресурсным характеристикам веб-приложений. Эмпирически верифицированные закономерности свидетельствуют, что приращение латентности всего на сто миллисекунд способно индуцировать падение коэффициента конверсии на величину порядка семи процентов. В обозначенных условиях оптимизация производительности undergoes фундаментальную трансформацию, эволюционируя из периферийной технической активности в стратегическую детерминанту конкурентоспособности хозяйствующих субъектов и интегральный индикатор качества пользовательского опыта. Ужесточение нормативных требований, манифестируемое, в частности, инициативой Google Core Web Vitals, дефакто институционализирует скорость отклика, интерактивность интерфейсов и визуальную стабильность макетов в качестве критериев, определяющих видимость цифрового продукта в поисковых системах, глубину вовлечённости аудитории и, в конечном счёте, монетизационный потенциал. Изложенный контекст актуализирует потребность в строгой систематизации и теоретическом осмыслении корпуса знаний, относящихся к инженерии производительности.

Аналитическое рассмотрение типовых дефектов, имманентно присущих процессам веб-разработки, эксплицирует наличие системных замедлений, порождаемых нерациональным управлением вычислительными и сетевыми ресурсами. Спектр подобных проблем простирается от некорректно сконфигурированных политик кэширования статических артефактов до генерации избыточных, семантически необоснованных запросов к подсистемам хранения данных. При масштабировании системы на аудиторию, характеризующуюся числом одновременных сеансов взаимодействия, превышающим десять тысяч, означенные архитектурные просчёты провоцируют нелинейный, лавинообразный рост нагрузки на серверные мощности, что, в свою очередь, влечёт эскалацию эксплуатационных расходов на хостинговую инфраструктуру на сорок-шестьдесят процентов и отток до пятидесяти трёх процентов пользователей вследствие неудовлетворительной отзывчивости интерфейсных компонентов.

Телеологическая установка настоящего пособия заключается в целенаправленном формировании у обучающихся системы компетенций в области квантификации и последующего улучшения показателей производительности, базирующейся на математическом моделировании сетевых задержек, анализе фундаментальных аппаратных ограничений и освоении формализованных инженерных методологий. Принципиальный акцент смещён в сторону достижения измеримых, верифицируемых результатов, а именно — обеспечения сокращения времени полной загрузки типового веб-приложения не менее чем на тридцать процентов, что подлежит объективной верификации посредством мониторинга динамики метрик Largest Contentful Paint и Time to Interactive.

Первая концептуальная задача излагаемого методического комплекса охватывает проблематику оптимизации клиентской подсистемы: от сравнительного анализа эффективности алгоритмов компрессии растровых данных (форматы WebP и AVIF) до имплементации механизмов отложенной, или ленивой, загрузки контента с задействованием программного интерфейса Intersection Observer. Иллюстративный материал включает аутентичные фрагменты программного кода для экосистемы React, наглядно эксплицирующие техники минимизации повторных циклов согласования виртуального DOM и подавления избыточных перерисовок компонентной иерархии. Вторая задача фокусируется на серверных методах, а именно: конфигурировании распределённого кэширования посредством сетей доставки контента (CDN), синтезе эффективных индексных структур в реляционных базах данных и реализации алгоритмов балансировки входящего трафика с использованием программных решений класса HAproxy или Nginx. Отдельное внимание уделяется количественной оценке влияния перечисленных мероприятий на результирующее время отклика через призму математических моделей теории очередей (в частности, марковской модели M/M/1), а также трансферу полученных теоретических конструктов в плоскость учебных проектных задач посредством разбора кейсов на платформе Node.js.

Содержательное наполнение пособия корреспондирует с требованиями федеральных государственных образовательных стандартов по направлениям подготовки 09.03.03 «Прикладная информатика» и 02.03.03 «Математическое обеспечение и администрирование информационных систем», в рамках которых оптимизация производительности трактуется как имманентно необходимый навык проектирования устойчивых, отказоустойчивых информационных систем. Структурирование материала осуществлялось с учётом профессио-

нальных компетенций, предусмотренных ФГОС, в аспекте разработки высоконагруженных приложений.

Научно-практическая значимость работы детерминирована лаконичностью формата изложения, органически сочетающего теоретико-методологический базис с набором аналитических инструментов: симуляторами сетевых задержек, вычислителями возврата на инвестиции (ROI) от оптимизационных мероприятий и формализованными шаблонами для проведения аудита производительности. Подобная архитектура пособия обеспечивает возможность непосредственной трансляции изложенных техник в реальные производственные проекты без приращения когнитивной нагрузки на обучающегося, гарантируя оперативное применение приобретённых знаний в профессиональной деятельности.

ГЛАВА 1. ФУНДАМЕНТАЛЬНЫЕ ОСНОВЫ ПРОИЗВОДИТЕЛЬНОСТИ ВЕБ-СИСТЕМ

Производительность веб-приложения, понимаемая как совокупность его временны́х и ресурсных характеристик в процессе обработки пользовательских запросов, является неотъемлемым атрибутом качества, имманентно присущим любой программной системе. В отличие от функциональных требований, описывающих, что система должна делать, производительность отвечает на вопрос, насколько эффективно она это делает. Настоящая глава закладывает теоретический и методологический фундамент дисциплины, вводя строгие определения, постулируя аксиоматику процесса оптимизации, анализируя её экономическую целесообразность и проводя таксономию узких мест, препятствующих достижению заданных показателей.

§1.1. Концептуальное определение оптимизации в программной инженерии

В контексте программной инженерии и, в частности, веб-разработки, термин «оптимизация» требует денормализации от своего общематематического значения — поиска глобального экстремума целевой функции. В реальных системах, характеризующихся высокой степенью неопределённости внешней среды (вариативность сетевых задержек, стохастический характер пользовательской активности, гетерогенность клиентских устройств), нахождение единого глобального оптимума является задачей некорректной или практически неразрешимой. Вследствие этого, под оптимизацией веб-приложений в рамках данного пособия предлагается понимать итеративный, измеримый и системный инженерный процесс, направленный на приближение ключевых показателей производительности (KPI) к целевым значениям, установленным бизнес-требованиями и ограничениями пользовательского опыта, при минимально возможных затратах вычислительных, временны́х и финансовых ресурсов.

Данная дефиниция имплицитно содержит несколько критически важных аспектов, отличающих профессиональную оптимизацию от интуитивных попыток «улучшить код».

Во-первых, это итеративность. Оптимизация не является разовым актом, выполняемым на финальной стадии проекта. Это перманентный циклический процесс, встроенный в жизненный цикл разработки. Цикл оптимиза-

ции (Рис. 1.1) включает фазы измерения текущего состояния, анализа собранных данных для идентификации узких мест, синтеза и имплементации решения, и последующей верификации, результаты которой становятся входными данными для следующей итерации. Такой подход, известный как «цикл Деминга» (Plan-Do-Check-Act) в применении к производительности, позволяет последовательно снижать энтропию системы.

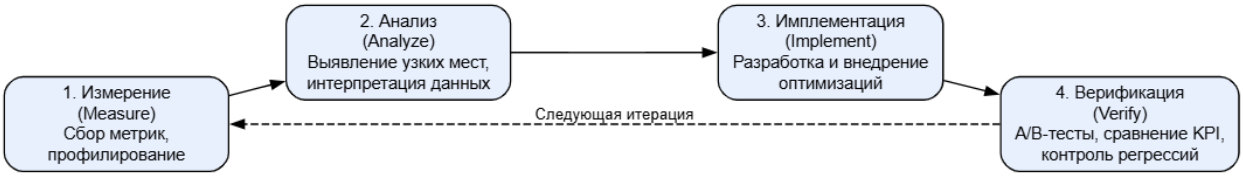

Рис.1.1. Циклическая модель процесса оптимизации производительности

Во-вторых, это измеримость. Как будет детально обосновано в параграфе 1.2, любые модификации системы, не подкреплённые объективными количественными данными об их эффекте, не могут классифицироваться как оптимизация. Субъективное ощущение «плавности» интерфейса или «быстроты» загрузки является необходимым, но недостаточным условием. Только квантификация таких показателей, как время до первого байта (TTFB), время до интерактивности (TTI) или кумулятивное смещение макета (CLS), позволяет перевести процесс улучшения из области искусства в область строгой инженерной дисциплины.

В-третьих, неотъемлемым атрибутом профессиональной оптимизации выступает системность, или холистический принцип рассмотрения объекта. Современное веб-приложение являет собой многоярусную распределённую гетерогенную систему, в которой клиентский браузерный агент, географически диспергированная инфраструктура доставки контента, конкретная реализация транспортного протокола, сервер прикладной логики, подсистема персистентного хранения данных и множество сторонних сервисов-сателлитов формируют топологически сложный, многосвязный граф функциональных и информационных зависимостей. Игнорирование данной сложности и проведение локальной, точечной оптимизации одного изолированного узла означенного графа, осуществляемой вне анализа её системных последствий для смежных и опосредованно связанных компонентов, способно продуцировать парадоксальный, контринтуитивный эффект — общую деградацию интегральных показателей производительности системы как целого.

Хрестоматийной иллюстрацией подобного контрпродуктивного исхода служит стратегия агрессивного бандлинга — объединения всей совокупности клиентских скриптов в монолитный файл значительного объёма, — мотиви-

рованная стремлением минимизировать количество HTTP-запросов. Данная тактика обладала несомненной рациональностью в эпоху доминирования протокола HTTP/1.1 с присущим ему фундаментальным ограничением в виде блокировки очереди запросов (Head-of-Line Blocking) и ограниченным пулом одновременных соединений к одному хосту. Однако в условиях функционирования поверх протокола HTTP/2 и, впоследствии, HTTP/3, реализующих нативное мультиплексирование множества независимых потоков данных в рамках единственного TCP- или QUIC-соединения, означенный подход становится откровенно дисфункциональным. Монолитный бандл элиминирует саму возможность гранулярной приоритизации загрузки критических субресурсов, блокирует механизмы избирательного кэширования на уровне отдельных чанков и нивелирует преимущества параллельной доставки, предоставляемые мультиплексированием. Следовательно, решение, оптимальное в одной технологической парадигме, трансформируется в антипаттерн при смене архитектурного контекста.

Резюмируя изложенное, системный подход постулирует необходимость анализа веб-приложения не как аддитивной совокупности автономных, изолированных модулей, но как целостного кибернетического организма, где каждый компонент находится в отношениях сложной коадаптации с остальными элементами, а эмерджентное поведение системы не редуцируется к сумме поведений её частей.

§1.2. Аксиоматический базис оптимизационного процесса: квантифицируемость, итеративность, холистичность

Для того чтобы процесс оптимизации производительности приобрёл свойства воспроизводимости, верифицируемости и прогнозируемости, он должен быть фундирован на совокупности фундаментальных постулатов, или аксиом, образующих его методологический каркас. Нарушение любого из этих базовых принципов с логической неизбежностью влечёт за собой диссипацию ресурсов — временны́х, вычислительных и финансовых — либо достижение результатов, знак которых противоположен изначально целеполагаемому. Формулируемые ниже аксиомы не являются произвольными нормативными установлениями; они суть логическая экспликация и развитие концептуальной дефиниции оптимизации как инженерной дисциплины, данной в предшествующем параграфе, и вытекают из самой природы распределённых веб-систем как объекта управляющего воздействия.

Аксиома 1: Примат измеримости над суждением (Axiom of Measurability).

«Невозможно оптимизировать то, что нельзя измерить». Этот постулат, являющийся адаптацией известного управленческого принципа лорда Кельвина, утверждает, что единственным легитимным основанием для внесения изменений в кодовую базу с целью повышения производительности является объективная, полученная инструментальным путём информация. Эмпирические догадки разработчика, каким бы большим ни был его опыт, не могут служить таким основанием в силу сложности и нелинейности поведения современных веб-систем. На практике это означает, что жизненный цикл любой оптимизационной задачи должен начинаться с этапа инструментации приложения и сбора массива данных, репрезентативно отражающего его поведение в целевых условиях эксплуатации.

Следствием данной аксиомы является требование к выбору метрик. Не все метрики равнозначны. Оптимизация «ложной» метрики, не коррелирующей с пользовательским опытом или бизнес-показателями, является пустой тратой ресурсов. Классический пример — оптимизация полного времени загрузки страницы (loadEventEnd) в ущерб времени отрисовки первого контента (FCP). Пользователь воспринимает приложение как быстрое, если видит контент, а не если завершилась загрузка скрытого аналитического скрипта в фоне. Поэтому выбор целевых метрик, релевантных конкретному приложению, является первым и важнейшим шагом.

Аксиома 2: Итеративность и принцип Парето-эффективного распределения усилий (Axiom of Iterative Improvement).

«Сложность веб-приложений делает практически невозможным нахождение и устранение всех узких мест за одну итерацию. Более того, устранение одного «бутылочного горлышка» неизбежно перемещает точку максимальной задержки в другой компонент системы.» Следовательно, оптимизация — это процесс последовательных улучшений, на каждом шаге которого решается задача поиска и устранения наиболее критичного в данный момент узкого места. Этот подход согласуется с принципом Парето: примерно 80% эффекта достигается за счёт устранения 20% проблем. Задача инженера по производительности — идентифицировать эти критически важные 20% и сконцентрировать усилия на них, а не распыляться на микрооптимизации, не дающие измеримого вклада в итоговый KPI.

Аксиома 3: Системный холизм и запрет на субоптимизацию (Axiom of Systemic Holism).

«Производительность конечного продукта является эмерджентным свойством всей системы, а не суммой производительностей её частей.» Субоптимизация — улучшение одного компонента, приводящее к ухудшению работы системы в целом, — является одной из самых распространённых ошибок. Как было показано в параграфе 1.1 на примере с бандлингом и HTTP/2, решение, эффективное в одном технологическом контексте, становится антипаттерном в другом. Другим примером может служить неограниченное кэширование всех ответов от API. Это может кардинально снизить нагрузку на сервер и улучшить время ответа для повторяющихся запросов, но одновременно привести к стагнации данных на клиенте и нарушению бизнес-логики, что в конечном счёте нанесёт больший ущерб, чем польза от ускорения. Системный подход требует анализа всего трейса запроса, от клика пользователя до ответа от базы данных и обратно, с учётом всех трансформаций и взаимодействий на этом пути.

§1.3. Экономика производительности: бизнес-ценность, конверсия и совокупная стоимость владения

Редуцирование производительности исключительно к разряду технических, инструментальных характеристик программного артефакта представляет собой концептуальную ошибку, существенно сужающую как обоснование инвестиций в оптимизационные мероприятия, так и понимание их стратегической роли в жизненном цикле цифрового продукта. Производительность надлежит трактовать как полноценный экономический фактор, оказывающий непосредственное, количественно измеримое и статистически верифицируемое воздействие на фундаментальные финансовые индикаторы хозяйствующего субъекта. Настоящий параграф посвящён экспликации двунаправленной каузальной связи между техническими метриками, с одной стороны, и бизнес-результатами, с другой.

Влияние на экономическую часть

Большой объем эмпирических исследований, выполненных как независимыми лабораториями, так и исследовательскими отделами крупнейших технологических корпораций, таких как Google, Amazon, Microfoft, Walmart и т.д., свидетельствует о наличии устойчивой, статистически значимой корреляции между скоростной характеристикой загрузки веб-страницы и коэффициентом конверсии — долей посетителей, совершивших целевое действие. Механизм, лежащий в основе данной зависимости, находит своё объяснение в концептуальном аппарате когнитивной психологии. Созданная техническими факторами задержка при слиянии с пользовательским интерфейсом

выполняет роль когнитивного прерывания т.е. она фрагментирует так называемое «потоковое состояние» ((flow state) — оптимальное психическое состояние полной вовлечённости в деятельность), — мультипликативно повышает когнитивную нагрузку на рабочую память пользователя и порождает фрустрацию, которая, в свою очередь, активирует поведенческие паттерны избегания и отказа от продолжения взаимодействия с интерфейсом. Таким образом, латентность интерфейса непосредственно транслируется в экономические потери, минуя промежуточные звенья субъективной оценки качества. В качестве графической иллюстрации на Рисунке 1.2 экспонируется типовая кривая, аппроксимирующая данную обратную зависимость и построенная на агрегированных кросс-отраслевых эмпирических данных, демонстрирующая выраженный нелинейный характер падения конверсии при выходе за пределы двухсекундного порога.

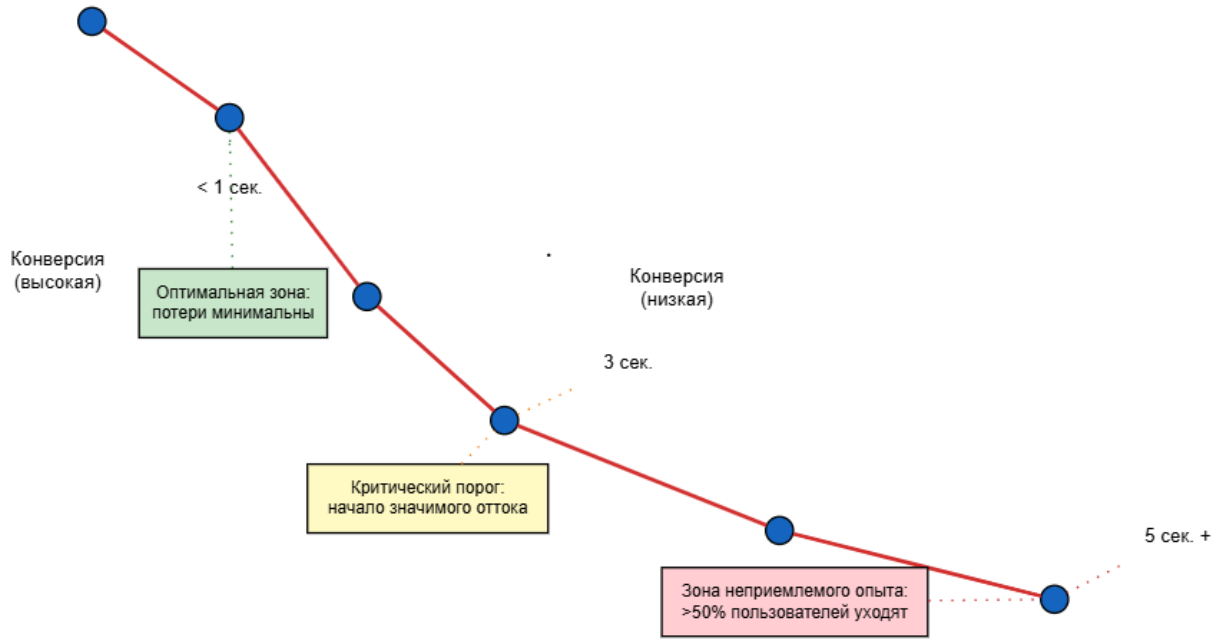

Рис.1.2. Качественная зависимость коэффициента конверсии от времени загрузки страницы. (Построено на основе агрегированных отраслевых данных)

Данные показывают, что даже незначительное, на первый взгляд, увеличение задержки (например, на 100 миллисекунд) может приводить к статистически значимому падению глубины просмотра и конверсии. Для высоконагруженных e-commerce платформ это напрямую транслируется в миллионные убытки. Таким образом, инвестиции в сокращение времени загрузки имеют чёткий и прогнозируемый возврат на инвестиции (ROI), который может быть рассчитан по формуле:

$$ROI_{per\ f} = \frac{C_{new} - C_{old} \cdot AOV \cdot T_{sessoins} - Cost_{opt}}{Cost_{opt}}$$

где C_{new} и C_{old} — коэффициенты конверсии после и до оптимизации соответственно, AOV (Average Order Value) — средний чек, $T_{sessoins}$ — количество целевых сессий за период, а $Cost_{opt}$ — совокупные затраты на проведение оптимизационных мероприятий.

Влияние на расходную часть: совокупная стоимость владения (TCO).

Помимо прямого вклада в генерацию валового дохода, методологически выверенная оптимизация производительности оказывает не менее существенное, хотя зачастую и недооцениваемое, воздействие на расходную часть финансовой модели цифрового продукта — а именно, на операционные издержки по содержанию и эксплуатации серверной инфраструктуры. Программный код, характеризуемый низкой степенью алгоритмической эффективности, генерация избыточных, семантически неоправданных запросов к системам управления базами данных, отсутствие либо неадекватная конфигурация многоуровневого кэширования и применение субоптимальных вычислительных стратегий — вся совокупность перечисленных факторов продуцирует неоправданно высокую степень утилизации критических аппаратных ресурсов: тактовой частоты центрального процессора, оперативной памяти и пропускной способности подсистемы ввода-вывода серверного оборудования. При масштабировании системы на аудиторию, исчисляемую десятками и сотнями тысяч конкурентных пользовательских сеансов, означенная ресурсная неэффективность проявляет свойство мультипликативности, экспоненциально увеличивая счета за потребление облачных вычислительных ресурсов в моделях с оплатой по факту использования (pay-as-you-go) либо требуя непропорционально высоких капитальных затрат на приобретение и обслуживание физического парка оборудования в on-premise сценариях.

Напротив, целенаправленная оптимизация серверной подсистемы, охватывающая, в частности, внедрение эшелонированных стратегий кэширования и глубинную реструктуризацию запросов к реляционным СУБД с применением соответствующих индексных структур, позволяет одному и тому же серверному инстансу, без приращения его физической мощности, обслуживать кратное количество запросов в единицу времени (RPS, Requests Per Second). Данное обстоятельство влечёт за собой два взаимосвязанных экономических следствия. Во-первых, происходит непосредственное снижение удельной стоимости обработки одного пользовательского запроса. Во-вторых, существенно отодвигается пороговое значение нагрузки, при достижении которого возникает объективная необходимость в горизонтальном

масштабировании — вводе в эксплуатацию дополнительных вычислительных узлов. В рамках парадигмы эластичного масштабирования, реализуемой современными облачными платформами, это означает, что в периоды пиковой, в том числе сезонной или маркетингово-обусловленной, активности система будет автоматически запрашивать у провайдера меньший объём дополнительных ресурсов, что транслируется в прямую экономию операционного бюджета. Таким образом, производительность должна концептуализироваться не как центр затрат, требующий перманентных финансовых вливаний, а как интегральный фактор, одновременно оптимизирующий и доходную, и расходную компоненты бизнес-модели цифрового продукта, повышая его маржинальность и инвестиционную привлекательность.

§ 1.4. Таксономия и этиология узких мест в распределённых веб-системах

Узкое место (bottleneck) в распределённой веб-системе — это компонент или процесс, пропускная способность или время отклика которого лимитирует общую производительность системы, вызывая задержки при обработке пользовательских запросов. Понимание природы и классификации узких мест необходимо для их целенаправленного поиска и устранения.

Узкие места можно классифицировать по нескольким основаниям.

1. По локализации в архитектурном стеке:

- Клиентские (Frontend): Связаны с парсингом и исполнением JavaScript, построением DOM и CSSOM, избыточными перерисовками (reflows/repaints), блокировкой основного потока выполнения (main thread blocking), загрузкой неоптимизированных медиаресурсов.
- Сетевые (Network): Обусловлены физическими ограничениями канала (RTT, пропускная способность), неэффективными протоколами (head-of-line blocking в HTTP/1.1), избыточным объёмом передаваемых данных, проблемами с резолвингом DNS.
- Серверные (Backend): Возникают из-за неэффективных алгоритмов обработки данных, блокировок в пуле потоков или цикле событий (event loop), проблем с управлением памятью, долгих сериализаций/десериализаций.
- Уровня данных (Data Layer): Связаны с отсутствием индексов в СУБД, блокировками на уровне строк или таблиц, избыточными соединениями (JOIN), N+1 проблемой запросов, медленными дисковыми операциями ввода-вывода.

2. По характеру возникновения:

- Ресурсные: Вызваны физическим исчерпанием ресурса. Например, утилизация CPU достигла 100%, и процессор не справляется с очередью задач. Или исчерпана пропускная способность сетевого канала.
- Контенционные: Возникают из-за конкуренции за разделяемый ресурс. Классический пример — блокировки в базе данных, когда несколько транзакций пытаются одновременно изменить одну и ту же запись. Другой пример — конкуренция за глобальную блокировку интерпретатора (GIL) в некоторых языковых средах.
- Конфигурационные: Являются следствием некорректной настройки окружения. Например, слишком маленький размер пула соединений с базой данных, неоптимальные таймауты ожидания, отключённый HTTP/2 на веб-сервере.
- Архитектурные: Фундаментальные ограничения, заложенные в дизайн системы. Например, синхронный обмен данными по сети в критическом пути запроса вместо асинхронного через очередь сообщений, что делает время отклика системы равным сумме задержек всех вызываемых сервисов.

Взаимодействие узких мест и метод последовательного исключения.

Важнейшим свойством системы с несколькими узкими местами является то, что они образуют цепочку. Устранение первого, самого «узкого» горлышка не приведёт к линейному улучшению, а лишь сместит точку насыщения ко второму по значимости ограничению. Этот процесс иллюстрируется теорией ограничений (Theory of Constraints) Элияху Голдратта, которая предлагает методологию из пяти шагов для управления ограничениями в системах. На Рис. 1.3 схематично показан процесс смещения узкого места после оптимизации.

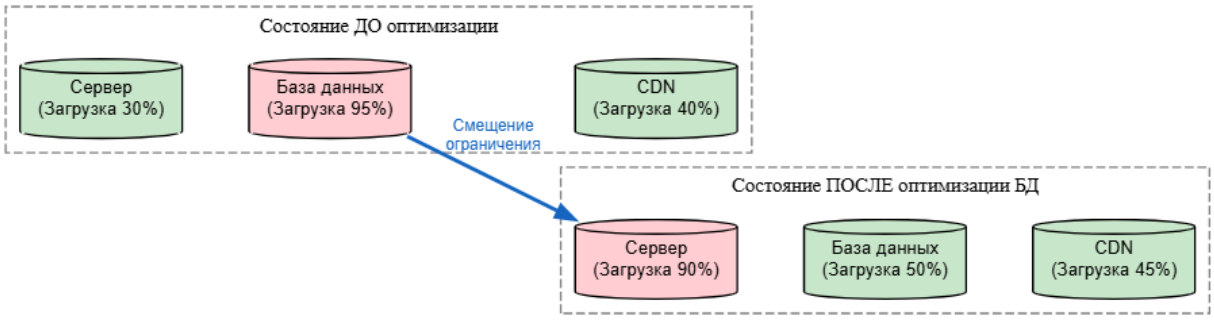

Рис. 1.3. Иллюстрация смещения узкого места в системе после оптимизации компонента

Практическим следствием из этого является **метод последовательного исключения**. Инженер по производительности должен идентифицировать текущее узкое место (например, с помощью построения трейсов запроса), устранить его, а затем перейти к следующему. Попытка оптимизировать всё одновременно не только неэффективна с точки зрения трудозатрат, но и лишает возможности верифицировать вклад каждого изменения.

§1.5. Типовые антипаттерны проектирования, приводящие к деградации

Завершающий параграф главы посвящён анализу распространённых ошибок проектирования и разработки, которые с высокой вероятностью приводят к проблемам с производительностью. Знание этих антипаттернов позволяет осуществлять их проактивное предотвращение на ранних этапах жизненного цикла, что значительно дешевле и эффективнее, чем реактивное исправление в production-среде. Данные антипаттерны являются логическим следствием нарушения аксиоматики, изложенной в параграфе 1.2.

Антипаттерн 1: Преждевременная оптимизация (Premature Optimization).

Этот антипаттерн, афористично сформулированный Дональдом Кнутом («Преждевременная оптимизация — корень всех зол»), заключается в усложнении кода ради потенциального выигрыша в производительности на участках, которые не являются узкими местами. Разработчик, основываясь на предположениях, а не на данных профилирования, применяет сложные, трудночитаемые и плохо поддерживаемые оптимизационные техники там, где они не нужны. Это нарушает Аксиому 1 (Примат измеримости). Сложный, запутанный код, в свою очередь, замедляет разработку и увеличивает вероятность внесения функциональных ошибок. Лекарством является строгое следование циклу «измерь-проанализируй-оптимизируй»: оптимизация должна проводиться только после того, как инструментальные средства доказали наличие проблемы в конкретном модуле.

Антипаттерн 2: Отсутствие ограничения ресурсоёмкости (Unbounded Resource Consumption).

Проектирование программных компонентов в отсутствие каких бы то ни было явных или имплицитных ограничений на объём потребляемых ими вычислительных ресурсов представляет собой прямой и наиболее короткий путь к возникновению каскадных отказов, когда локальный инцидент в одной подсистеме запускает цепную реакцию деградации во всех смежных. Ниже приведены некоторые проявления данного антипаттерна:

- Неограниченная выборка данных (Unbounded Query): Формирование SQL-запроса вида SELECT * FROM table без спецификации лимитирующих директив LIMIT и OFFSET, а также без явного перечисления релевантных столбцов. Подобный запрос способен результировать в материализацию на стороне сервера приложений реляционного отношения, содержащего миллионы кортежей, каждый из которых включает десятки атрибутов, в ситуации, когда для реализации бизнес-логики фактически требуется лишь незначительное подмножество записей. Последствия носят триединый характер: во-первых, происходит исчерпание доступной оперативной памяти серверного процесса (Out-of-Memory), во-вторых, сетевой канал между сервером приложений и СУБД забивается избыточным объёмом передаваемых данных, в-третьих, наступает полная функциональная деградация сервиса вплоть до его недоступности. Императивным проектным требованием выступает обязательное применение пагинации, предикативной фильтрации на стороне СУБД и эксплицитное указание требуемых столбцов в секции SELECT.
- Доставка полноформатных растровых изображений: Трансляция пользовательскому агенту графических файлов в их нативном, исходном разрешении (например, 4000×3000 пикселей) для их последующего отображения в области видимости, зарезервированной под миниатюру размером 100×100 пикселей, в составе, скажем, новостной ленты. Данная практика влечёт передачу по сетевому каналу, в том числе по каналам с высокой задержкой (высоколатентным) и тарифицируемым мобильным соединениям, мегабайт балластных, семантически избыточных данных, что мультипликативно увеличивает как время полной загрузки страницы, так и объём потреблённого пользователем трафика. Инженерно корректным решением является имплементация стратегии адаптивных изображений, предусматривающей предварительную либо динамическую (на лету, средствами Image CDN) генерацию множества вариантов файла, релевантных целевым размерам и пиксельной плотности клиентских устройств.
- Рекурсивные или иерархически вложенные структуры без лимитирования глубины: Ситуация, в которой клиентский запрос, например, на языке GraphQL, потенциально способен специфицировать бесконечно рекурсивную вложенность (комментарии к комментариям, теги к тегам и т.п.) без каких-либо предохранительных механизмов на серверной стороне. Исполнение подобного запроса способно привести к полному исчерпанию вычислительных ресурсов сервера и его аварий-

ной остановке. Необходимой контрмерой является внедрение анализатора стоимости запроса (Query Cost Analysis) с жёстко заданным лимитом максимальной глубины вложенности и общего веса запроса.

Антипаттерн 3: Пренебрежение асинхронной природой сетевого взаимодействия (Sync-over-Async).

Одной из наиболее глубоких и системных ошибок архитектурного проектирования распределённых систем выступает выстраивание цепочек межсервисного взаимодействия исключительно на базе синхронных, блокирующих вызовов, помещённых непосредственно в критический путь обработки пользовательского запроса. В рамках подобной парадигмы, если сервис A для формирования результирующего ответа конечному потребителю вынужден синхронно ожидать завершения обработки запроса сервисом B, который, в свою очередь, находится в аналогичной блокирующей зависимости от сервиса C, то результирующее время отклика всей композиции оказывается равным не максимуму, а сумме индивидуальных задержек и таймаутов каждого из последовательно вызываемых компонентов. Более того, любой инцидент — будь то транзиентный сбой, временная деградация или полный отказ — в любом из downstream-сервисов (B или C) немедленно, без какой-либо буферизации или амортизации, пропагируется вверх по цепочке вызовов, индуцируя каскадную деградацию всех вышележащих, в том числе непосредственно клиенто-ориентированных, сервисов. Данный антипаттерн вступает в прямое противоречие с Аксиомой 3 (Системный холизм), поскольку порождает жёсткие, хрупкие и неамортизированные связи между компонентами, делая систему в целом значительно менее устойчивой, чем любая из её частей по отдельности. Архитектурным разрешением означенной проблемы служит переход к асинхронным, событийно-управляемым (event-driven) моделям взаимодействия, предполагающим внедрение промежуточного транспортного слоя в виде распределённых очередей сообщений (Apache Kafka, RabbitMQ) либо применение реактивных декларативных паттернов (Reactive Streams, Project Reactor) везде, где семантика бизнес-процесса допускает отказ от строгой синхронности.

Антипаттерн 4: Индискриминантное кэширование в отсутствие формализованной стратегии инвалидации.

Кэширование, как было сказано в предшествующем изложении, представляет собой один из наиболее эффективных инструментов повышения производительности, однако его бездумная, тотальная аппликация ко всем без разбора типам данных способна привести к последствиям, характеризуе-

мым как катастрофические. Профессиональное сообщество разработчиков на протяжении десятилетий циркулирует афористичное наблюдение, приписываемое Филу Карлтону: «Существуют две фундаментально сложные проблемы в Computer Science: инвалидация кэша, именование переменных и ошибка смещения на единицу». За остроумной формой этого высказывания скрывается глубокая инженерная истина. Установка неоправданно продолжительного времени жизни кэшируемой сущности (TTL, Time to Live) либо, что ещё более критично, полное отсутствие механизмов детерминированной, событийно-триггерируемой инвалидации при модификации исходных данных в мастер-хранилище с неизбежностью приводит к тому, что конечные пользователи начинают наблюдать устаревшую, более не соответствующую действительности, а в отдельных случаях — откровенно некорректную информацию. Для ряда предметных областей — финансовый и банковский сектор, медицинские информационные системы, оперативные новостные ленты — подобная стагнация данных является категорически неприемлемой и может повлечь за собой не только репутационные, но и юридические последствия. Из изложенного следует императив: стратегия кэширования не должна быть надстройкой, апостериорно добавляемой к готовой модели данных. Она обязана проектироваться синхронно с разработкой самой модели бизнес-логики, и специфицировать не только перечень кэшируемых сущностей и длительность их хранения, но и исчерпывающий перечень условий и триггеров, при наступлении которых кэшированная копия должна быть признана недействительной и принудительно эвакуирована из кэш-хранилища.

Изучение и осмысление этих фундаментальных основ создаёт необходимую базу для перехода к практическим аспектам метрологии производительности, которые будут детально рассмотрены далее. Понимание того, что мы измеряем, зачем мы это делаем и какие системные ошибки нас подстерегают, является обязательным условием для проведения эффективной оптимизации.

ВЫВОДЫ

Резюмируя содержание первой главы, представляется необходимым зафиксировать фундаментальный сдвиг в понимании природы оптимизации веб-приложений, который был в ней обоснован. Оптимизация есть не совокупность разрозненных тактических приёмов, а имманентно присущая процессу программной инженерии методологическая рамка, базирующаяся на трёх нередуцируемых аксиомах: примате измеримости над субъективным суждением, итеративности улучшений, направляемых принципом Парето, и системном холизме, запрещающем субоптимизацию. Нарушение любой из

этих аксиом с неизбежностью переводит деятельность из плоскости инженерии в плоскость неверифицируемых, а следовательно, и невоспроизводимых манипуляций.

Проведённый анализ экономики производительности вскрыл двустороннюю связь между техническими метриками и финансовыми результатами. Производительность была эксплицирована не как центр затрат, а как фактор, одновременно воздействующий и на доходную часть (через конверсию и удержание), и на расходную (через снижение совокупной стоимости владения инфраструктурой). Формализация этой связи через расчёт возврата на инвестиции переводит обсуждение производительности с языка технических предпочтений на язык бизнес-обоснований, что является необходимым условием для выделения ресурсов на оптимизационные мероприятия.

Таксономия узких мест и анализ типовых антипаттернов, завершающие главу, создают диагностический инструментарий, позволяющий инженеру не просто констатировать наличие проблемы, но и классифицировать её природу — ресурсную, контенционную, конфигурационную или архитектурную. Понимание того, что узкие места образуют цепочку, подчиняющуюся логике Теории ограничений, и что их устранение есть процесс последовательного смещения точки максимальной задержки, является ключевым метакогнитивным результатом, предотвращающим хаотичную и неэффективную деятельность.

Вопросы для самопроверки

1. Дайте строгое определение понятию «оптимизация веб-приложений» в контексте программной инженерии. В чём его принципиальное отличие от общематематической трактовки оптимизации как поиска глобального экстремума?
2. Раскройте содержание трёх ключевых аспектов оптимизации: измеримости, итеративности и системности. Почему нарушение любого из них приводит к неэффективному расходованию ресурсов?
3. Сформулируйте Аксиому примата измеримости над суждением. Приведите конкретный пример из практики веб-разработки, когда её игнорирование привело к отрицательному результату.
4. Что понимается под термином «субоптимизация»? Проиллюстрируйте это понятие на примере агрессивного бандлинга скриптов в эпоху HTTP/2.

5. Опишите механизм влияния времени загрузки страницы на коэффициент конверсии. Какие когнитивные факторы лежат в основе этой зависимости?
6. Приведите формулу расчёта возврата на инвестиции (ROI) от оптимизации производительности. Поясните экономический смысл каждой переменной.
7. Дайте определение понятию «узкое место» (bottleneck) в распределённой системе. Проведите классификацию узких мест по локализации в архитектурном стеке.
8. В чём заключается метод последовательного исключения при работе с узкими местами? Как он соотносится с Теорией ограничений Э. Голдратта?
9. Охарактеризуйте антипаттерн «Преждевременная оптимизация». Приведите исторический контекст его возникновения и практические последствия.
10. Раскройте сущность антипаттерна «Отсутствие ограничения ресурсоёмкости». Какие механизмы должны быть предусмотрены на этапе проектирования для его предотвращения?

Задания на количественный анализ

Задача 1.1. Расчёт потерь конверсии.

Время загрузки страницы интернет-магазина составляет 4.2 секунды. После оптимизации его планируется сократить до 1.8 секунды. Известно, что каждая секунда задержки снижает конверсию на 7%. Текущий коэффициент конверсии — 3.2%, средний чек — 4500 руб., ежемесячная посещаемость — 120 000 целевых сессий. Определите ожидаемый прирост ежемесячной выручки после оптимизации.

Задача 1.2. Оценка ROI оптимизации.

Затраты на проведение оптимизационных мероприятий составили 350 000 руб. В результате среднее время загрузки сократилось с 5.1 с до 2.3 с. Исходный коэффициент конверсии — 2.1%, средний чек — 3200 руб., ежемесячный трафик — 80 000 сессий. Считая зависимость конверсии от времени линейной с коэффициентом 7% потерь на секунду, вычислите срок окупаемости инвестиций в месяцах.

Задача 1.3. Анализ смещения узкого места.

Система состоит из трёх последовательных компонентов: A (время обработки 40 мс), B (время обработки 120 мс), C (время обработки 60 мс). Общее время ответа системы составляет 220 мс. После оптимизации компонента B его время обработки сократилось до 30 мс. Определите: а) новое общее время ответа системы; б) компонент, ставший новым узким местом; в) на сколько процентов улучшилось общее время ответа.

Задача 1.4. Принцип Парето в оптимизации.

Профилирование выявило четыре проблемы, вносящие вклад в общую задержку: P1 — 450 мс (требует 2 человеко-дня на исправление), P2 — 300 мс (5 человеко-дней), P3 — 180 мс (1 человеко-день), P4 — 70 мс (8 человеко-дней). Общая задержка составляет 1000 мс. Используя принцип Парето, определите оптимальную последовательность устранения проблем для достижения 80% возможного улучшения с минимальными трудозатратами.

Задача 1.5. Оценка совокупной стоимости владения (TCO).

Серверный кластер из 5 узлов обслуживает 500 RPS при средней утилизации CPU 75%. Стоимость одного узла в месяц — 12 000 руб. После оптимизации кода утилизация CPU снизилась до 45% при той же нагрузке. Определите: а) сколько узлов можно вывести из эксплуатации без потери производительности; б) годовую экономию на инфраструктуре.

Задача 1.6. Расчёт потерь от отказов.

Вероятность отказа пользователя от сайта $P(bounce)$ связана с временем загрузки T (в секундах) эмпирической зависимостью:

$$P(bounce) = 1 - e^{-0.3 \cdot T}$$

Определите: а) вероятность отказа при времени загрузки 2 с и 6 с; б) на сколько процентных пунктов снизится вероятность отказа при сокращении времени загрузки с 5 с до 2 с.

Задача 1.7. Анализ контенционного узкого места.

Сервер обрабатывает запросы в один поток. Интенсивность поступления запросов $\lambda = 15$ заявок/с, интенсивность обслуживания $\mu = 20$ заявок/с.

Определите: а) коэффициент загрузки системы ρ; б) во сколько раз увеличится среднее время ответа, если интенсивность поступления возрастёт до 18 заявок/с.

Задача 1.8. Оценка эффекта от кэширования.

Без кэширования каждый запрос к базе данных занимает 80 мс. Внедрение кэша с hit-ratio 75% позволяет обслуживать попадания за 2 мс. Определите среднее время обработки запроса после внедрения кэша и относительное ускорение.

Задача 1.9. Расчёт потерь от блокировки рендеринга.

Критический ресурс (CSS-файл) загружается с CDN за 120 мс, но из-за высокой задержки DNS-резолвинга фактическое время начинает составлять 350 мс. На сколько миллисекунд возрастает время до First Contentful Paint (FCP), если этот CSS блокирует рендеринг?

Задача 1.10. Сравнение стратегий загрузки.

Страница загружает три скрипта: A (150 КБ), B (80 КБ), C (200 КБ). При последовательной загрузке (HTTP/1.1) с RTT = 100 мс и пропускной способностью 1 МБ/с общее время составляет сумму времён загрузки каждого. При параллельной загрузке (HTTP/2) время определяется самым большим файлом. Определите выигрыш от перехода на HTTP/2 в миллисекундах.

ГЛАВА 2. МЕТРОЛОГИЯ ПРОИЗВОДИТЕЛЬНОСТИ: МЕТРИКИ, МОДЕЛИ И ИНСТРУМЕНТАРИЙ

Производительность веб-приложения, как было установлено в Главе 1, является эмерджентным свойством сложной распределённой системы. Её изучение, контроль и улучшение невозможны без развитого метрологического аппарата — совокупности методов, показателей и инструментов, обеспечивающих получение объективной, воспроизводимой и интерпретируемой информации о поведении системы. Настоящая глава посвящена систематизации знаний в области измерения производительности: от таксономии метрик и их семантического анализа до математических моделей, лежащих в основе сетевых и вычислительных ограничений, и инструментальных средств, позволяющих реализовать на практике принцип «без метрик нет оптимизации».

§2.1. Таксономия метрик: лабораторные, полевые и синтетические измерения

Многообразие аспектов производительности веб-приложений и условий их функционирования обусловливает необходимость классификации измерительных подходов. Традиционное деление на «лабораторные — поле-

вые» метрики, хотя и является базовым, но недостаточно для полного охвата современных практик. Предлагается расширенная таксономия, включающая три категории измерений, различающихся по степени контролируемости среды, репрезентативности данных и целям применения.

Лабораторные (синтетические) измерения реализуются в жёстко регламентированной, контролируемой инструментальной среде, где инженер-исследователь наделён полномочиями детерминированно специфицировать и фиксировать всю совокупность релевантных параметров: конфигурацию аппаратного обеспечения (тип и тактовую частоту центрального процессора, объём оперативной памяти), характеристики сетевого соединения (пропускную способность, латентность, уровень пакетных потерь), версии операционной системы и браузерного агента, а также точную последовательность шагов пользовательского сценария. Данный методологический подход обеспечивает предельно достижимую степень воспроизводимости получаемых результатов, что делает его безальтернативным инструментом для задач регрессионного тестирования производительности и сравнительного (компаративного) анализа — например, при сопоставлении эффективности двух последовательных ревизий программного кода или конкурирующих технологических фреймворков.

Конституирующей характеристикой лабораторных измерений выступает их принципиальная детерминированность в границах наперёд заданной конфигурации. Профильные инструментальные средства, такие как Google Lighthouse, платформа WebPageTest (функционирующая в режиме синтетического тестирования) или автоматизированные скрипты, построенные на базе протокола Chrome DevTools Protocol с использованием библиотеки Puppeteer, предоставляют возможность многократного, циклического воспроизведения идентичного пользовательского сценария и получения на выходе статистически гомогенной выборки метрик. Данное свойство, в свою очередь, позволяет изолировать и квантифицировать влияние конкретного, точечного изменения в кодовой базе или конфигурационном файле на результирующие показатели — операция, принципиально неосуществимая в условиях реальной эксплуатации с присущей ей имманентной, неконтролируемой вариативностью внешних факторов. Вместе с тем, имманентным ограничением лабораторного подхода является его потенциальная «стерильность» и, как следствие, известная степень оторванности от аутентичного пользовательского опыта: синтетический тест может не учитывать такие феномены, как троттлинг (принудительное снижение тактовой частоты) центрального процессора мобильного устройства при достижении критических

температур (thermal throttling), стохастическую природу электромагнитных помех в беспроводных сетях или реальные, зачастую иррациональные, поведенческие паттерны живой аудитории.

Полевые (реальные) измерения (Real User Monitoring, RUM), напротив, реализуются посредством пассивного либо активного сбора телеметрических данных о производительности непосредственно на клиентских устройствах, у конечных пользователей в процессе их естественного, недетерминированного взаимодействия с приложением. Данный подход обеспечивает максимально возможную степень репрезентативности (т.е. точного отражения свойств изучаемого приложения или интерфейса), т.к. фиксирует подлинный, не модельный, опыт аудитории во всей его гетерогенности: от флагманских аппаратов, функционирующих в высокоскоростных сетях, до бюджетных смартфонов, подключённых к слабым сетям с высокой латентностью (задержками) и значительными пакетными потерями. Массивы RUM-данных позволяют делать явными те проблемы, которые практически не поддаются адекватному лабораторному моделированию: влияние географической дистанции между пользователем и точками присутствия CDN, сезонные и суточные (диурнальные, дневные) флуктуации (любые случайные отклонения или колебания значений) производительности, а также статистически значимые корреляции метрик с конкретными аппаратными платформами или версиями браузеров.

Слабой стороной полевых (реальных) данных является их высокая степень зашумлённости (т.е. при сборе данных присутствует большое количество системного мусора) и, как следствие, высокая сложность для дальнейшей интерпретации. На результирующие метрики оказывает одновременное воздействие множество неконтролируемых и зачастую латентных факторов, что делает обязательным применение развитого аппарата математической статистики для элиминации шума, выявления статистически значимых трендов и детекции аномалий. Кроме того, сама процедура сбора RUM-данных предполагает внедрение в программный код приложения специализированных скриптов-маяков (beacons), что потенциально способно оказывать паразитное влияние на измеряемую производительность и, следовательно, должно осуществляться с неукоснительным соблюдением принципа минимальной ресурсоёмкости и асинхронности исполнения.

Синтетические измерения в продуктовом окружении (Synthetic Monitoring) занимают промежуточное положение между двумя вышеназванными категориями. Они предполагают выполнение заранее запрограммированных скриптов, имитирующих поведение пользователя, но не в изолиро-

ванной лабораторной среде, а из географически распределённых точек присутствия против реально функционирующего production-окружения. Этот подход сочетает воспроизводимость лабораторных тестов с учётом реальной сетевой топологии и состояния production-инфраструктуры. Synthetic Monitoring незаменим для контроля доступности сервиса (uptime monitoring) и верификации соблюдения Service Level Agreements (SLA) по ключевым географическим регионам.

Взаимосвязь и взаимодополняемость данных категорий иллюстрируется Рисунком 2.1. Эффективная стратегия измерения производительности должна строиться на синергии всех трёх подходов, где данные каждого типа используются для решения специфических задач: лабораторные — для отладки и регрессионного контроля, полевые — для понимания реального пользовательского опыта и приоритизации оптимизаций, синтетические в продакшене — для оперативного мониторинга и алертинга.

Рис.2.1. Таксономия метрик производительности и их взаимосвязь

§2.2. Семантический анализ метрик пользовательского опыта (Core Web Vitals)

Переход от измерения абстрактных технических параметров к оценке качества пользовательского опыта потребовал разработки семантически насыщенных метрик, отражающих не просто факт завершения того или иного события, а его восприятие человеком. Инициатива Google Core Web Vitals, включённая в алгоритмы ранжирования поисковой выдачи, формализовала три ключевые метрики, которые стали отраслевым стандартом. Данный параграф посвящён их глубинному семантическому анализу, выходящему за рамки простого определения.

Largest Contentful Paint (LCP): перцептивная готовность основного контента.

Метрика LCP измеряет время, прошедшее от начала загрузки страницы до момента рендеринга самого крупного видимого элемента контента в области просмотра (viewport). В отличие от своих предшественников (First

Contentful Paint, Speed Index), LCP фокусируется не на первом появлении любого контента, а на моменте, когда пользователь с высокой вероятностью может воспринять страницу как «полезную» и «загруженную». Этим элементом может быть изображение-заставка, крупный текстовый блок или фоновое видео.

Семантика LCP тесно связана с концепцией перцептивной завершённости. Психологические исследования в области HCI показывают, что пользователи оценивают скорость загрузки не по окончанию фоновой активности, а по моменту стабилизации основного визуального содержимого. Не превышающее отметку в две с половиной секунды значение LCP с высокой степенью статистической достоверности коррелирует с положительной пользовательской оценкой опыта взаимодействия, квалифицируемого как «быстрый» и «отзывчивый». Напротив, преодоление четырёхсекундного рубежа переводит опыт в категорию «неудовлетворительный» (poor), сопряжённый с выраженным ростом вероятности отказа от дальнейшего взаимодействия и формирования негативной поведенческой реакции. Достижение целевых значений данной метрики требует имплементации не единичного, точечного вмешательства, а комплексной, многоуровневой оптимизационной стратегии, охватывающей три принципиально различных, но взаимообусловленных домена: во-первых, сокращение времени формирования первого байта ответа (TTFB) на стороне серверной инфраструктуры; во-вторых, оптимизацию канала доставки контента, включая задействование географически распределённых сетей (CDN) и применение эффективных алгоритмов транспортного сжатия; в-третьих, минимизацию задержек на этапе клиентского рендеринга, что подразумевает элиминацию блокирующих синтаксический анализатор скриптов и таблиц стилей из критического пути отрисовки.

First Input Delay (FID) и Interaction to Next Paint (INP): от дискретной задержки реакции к перманентной отзывчивости.

Метрика First Input Delay (FID) квантифицирует временной промежуток между моментом инициации первого пользовательского взаимодействия — будь то клик, тап по сенсорному экрану или нажатие клавиши — и моментом, когда браузерный агент оказывается в состоянии приступить к обработке соответствующего событийного обработчика. По своей природе FID является косвенным индикатором степени загруженности основного потока выполнения (main thread) в момент начала жизненного цикла страницы, т.е. фазы интерактивности. Однако данной метрике присуще фундаментальное, неустранимое ограничение: она фиксирует исключительно первое взаимодей-

ствие и исключительно временную задержку до начала обработки, полностью игнорируя последующие акты интеракции и визуальный результат т.е. ответы интерфейса на дальнейшие действия пользователя.

На смену FID приходит более совершенная и насыщенная метрика Interaction to Next Paint (INP), которая оценивает отзывчивость интерфейса на всём протяжении жизненного цикла страницы, т.е. от момента её полной загрузки до завершения работы пользователя. INP агрегирует данные о всех зафиксированных взаимодействиях (кликах, тапах, нажатиях клавиш) и в качестве итогового значения фиксирует наихудшую, с точки зрения пользователя, задержку от начала события и до следующего кадра визуальной отрисовки. Таким образом, INP осуществляет методологический сдвиг в оценке интерактивности: от анализа изолированного, единичного события к анализу непрерывной, персистентной отзывчивости. Семантически данная метрика значительно ближе к субъективному пользовательскому восприятию «подтормаживаний» и «залипаний» интерфейса: даже если первое нажатие кнопки было обработано практически мгновенно, регулярные микрофризы при скроллинге или при вводе текста в форму формируют у пользователя целостное негативное впечатление о качестве продукта.

Cumulative Layout Shift (CLS): визуальная стабильность как фактор доверия и предиктор ошибочных действий.

Метрика Cumulative Layout Shift (CLS) предоставляет количественную оценку суммарного, непреднамеренного и не инициированного пользователем смещения видимых элементов компоновки страницы в течение всего периода её существования. В отличие от рассмотренных выше темпоральных метрик, оперирующих единицами времени, CLS измеряет пространственную нестабильность макета. Вычислительный механизм CLS базируется на мультипликативной свёртке двух безразмерных величин: доли области просмотра (viewport), затронутой смещением (impact fraction), и относительного расстояния, на которое сместился элемент, нормированного на высоту области просмотра (distance fraction).

Семантически CLS отражает не просто эстетическое неудобство или дискомфорт восприятия, но фактор, непосредственно влияющий на доверие пользователя к интерфейсу и безопасность совершаемых им действий. Внезапное, недетерминированное смещение контента способно спровоцировать ошибочные моторные акты: пользователь, целенаправленно перемещающий курсор или палец к кнопке «Отмена» для прерывания нежелательной операции, в последнее мгновение обнаруживает, что под его указателем вслед-

ствие сдвига макета оказалась кнопка «Подтвердить» или «Купить». Подобные инциденты, классифицируемые как «тёмные паттерны» по факту, даже если они не были злонамеренно спроектированы, подрывают доверие к цифровому продукту и провоцируют остру фрустрацию. К числу основных этиологических факторов высокого CLS относятся: динамически инжектируемые рекламные блоки, встраиваемые без предварительного резервирования геометрического пространства; растровые изображения, специфицированные в разметке без явного указания размеров (атрибуты width и height); асинхронно загружаемые веб-шрифты, вызывающие феномен сдвига текста при подстановке (FOUT — Flash of Unstyled Text, или FOIT — Flash of Invisible Text). Целевое, эталонное значение CLS, согласно отраслевым стандартам, не должно превышать пороговой величины в 0.1.

Взаимосвязь Core Web Vitals и их влияние на пользовательское восприятие схематично представлены на Рисунке 2.2.

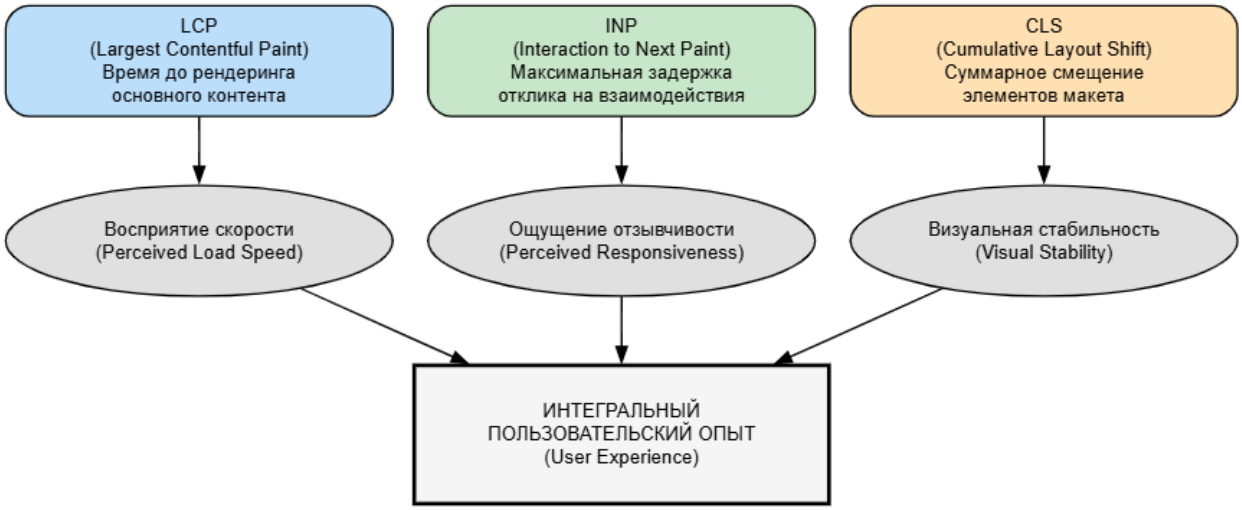

Рис.2.2. Семантическая модель Core Web Vitals и их влияния на пользовательский опыт

§2.3. Инструментальные средства мониторинга и профилирования

Эффективная реализация принципа измеримости, постулированного в Главе 1, невозможна без владения арсеналом инструментальных средств. Данный параграф представляет систематизированный обзор инструментов, классифицированных по их месту в жизненном цикле разработки и типу решаемых задач. Особое внимание уделяется не просто перечислению, а анализу методологических подходов, лежащих в основе их работы.

Инструменты этапа разработки (Development-Time Tools).

На этапе написания и отладки кода критически важно иметь возможность оперативно выявлять проблемы производительности, не дожидаясь развёртывания на тестовом окружении. Основным инструментом здесь выступают встроенные в браузеры средства разработчика (DevTools). Вкладка Performance в Chrome DevTools предоставляет детальную запись временной шкалы работы приложения, включая пламенные диаграммы (flame charts) исполнения JavaScript, очереди микро- и макрозадач, этапы построения DOM и CSSOM, операции компоновки (layout) и отрисовки (paint). Профилировщик JavaScript позволяет выявлять «горячие» функции, на выполнение которых тратится непропорционально большое время. Вкладка Network анализирует водопадную диаграмму (waterfall) загрузки ресурсов, визуализируя очереди, задержки и время загрузки каждого файла, что позволяет диагностировать проблемы блокировки рендеринга скриптами или избыточный вес ресурсов.

Для серверной разработки (в частности, на платформе Node.js) незаменимыми являются встроенный профайлер --inspect и clinic.js. Они позволяют детально анализировать утилизацию CPU, потребление памяти и паттерны асинхронного выполнения, выявляя блокировки цикла событий (Event Loop), утечки памяти и неоптимальные операции ввода-вывода.

Инструменты непрерывной интеграции (CI/CD Tools).

Для предотвращения регрессий производительности при внесении изменений в кодовую базу инструменты измерения должны быть интегрированы в конвейер CI/CD. Автоматизированные аудиты с помощью Lighthouse CI или Sitespeed.io позволяют на каждом pull request'е выполнять синтетические тесты и сравнивать полученные метрики с эталонными значениями или результатами предыдущих сборок. Настройка бюджета производительности (performance budget) — например, ограничение размера JavaScript-бандла или времени LCP — и автоматический алерт при его превышении является лучшей практикой, позволяющей «сдвинуть влево» (shift left) контроль производительности.

Инструменты эксплуатационного мониторинга (Production Monitoring).

В production-окружении задачи мониторинга смещаются от детального профилирования к агрегации и анализу потоковых данных от реальных пользователей. Системы класса Real User Monitoring (*например, Grafana Faro, Datadog RUM, Sentry Performance*) внедряются в приложение в виде SDK и собирают с клиентских устройств метрики Core Web Vitals, данные о време-

ни ответа API, ошибках и кастомные бизнес-показатели. Серверная телеметрия агрегируется посредством специализированных решений класса Application Performance Management (APM), к числу которых относятся, в частности, OpenTelemetry, New Relic и Elastic APM. Данные платформы обеспечивают реализацию механизма распределённой трассировки запросов (distributed tracing), позволяющего проследить полный путь пользовательской заявки сквозь всю топологию микросервисного ландшафта, а также централизованное структурированное логирование. Ключевым конкурентным преимуществом современных APM-систем является их способность к синтезу унифицированного, сквозного трейса, который неразрывно связывает событие, инициированное на клиентской стороне, со всей цепочкой серверных операций, включая терминальные запросы к подсистемам персистентного хранения данных. Подобная сквозная корреляция радикально, на порядки, сокращает временные затраты на локализацию первопричин узких мест и диагностику инцидентов производительности.

§2.4. Конструирование системы ключевых показателей эффективности на базе организационных целей

Метрики, подвергнутые анализу в предшествующих параграфах, по своей природе являются техническими индикаторами — они характеризуют поведение системы, но не эксплицируют его ценность для хозяйствующего субъекта. Их прагматическая значимость для бизнеса и команды разработки подвергается мультипликативному усилению в том случае, когда разрозненные технические показатели интегрируются в целостную, иерархически организованную систему ключевых показателей эффективности (Key Performance Indicators, KPI), жёстко привязанную к стратегическим целям организации. Процесс построения такой системы представляет собой процедуру трансляции качественных бизнес-целей на язык конкретных, измеримых, количественно верифицируемых технических порогов.

Начальным шагом данного процесса является определение целевых уровней обслуживания (Service Level Objectives, SLO). SLO представляет собой специфицированное числовое значение контролируемой метрики либо допустимый диапазон значений, который команда-разработчик принимает на себя обязательство поддерживать в течение оговоренного временно́го интервала. Здесь принципиально важным является следующее замечание: использование среднего арифметического для временны́х метрик, категорически противопоказано, т.к. данная статистика обладает свойством маскировать «тяжёлый хвост» распределения, — т.е. те самые наихудшие значения, которые и формируют негативный пользовательский опыт на наименее произво-

дительных устройствах (устаревшие версии ОС, такие как windows xp, vista, 7 и пр., старые мобильные телефоны, объем ОЗУ которых едва хватает для работы самой ОС) или в наиболее деградированных сетевых условиях (при медленном сетевом подключении, например менее 10 Мбит/сек и т.д.).

Следующим (вторым) шагом следует формализация соглашений об уровне обслуживания (Service Level Agreements, SLA). В отличие от SLO, носящих характер внутреннего, инженерного ориентира, SLA представляет собой юридически обязывающий контракт с бизнес-подразделениями или внешними заказчиками, эксплицитно определяющий санкции — как правило, финансовые — за систематическое несоблюдение оговоренных показателей. Нарушение SLO классифицируется как инцидент, инициирующий инженерное расследование. Нарушение же SLA влечёт за собой ощутимые бизнес-последствия, что радикально повышает ставки и требует выстраивания многоуровневой системы мониторинга и алертинга.

Третьим, завершающим шагом является непосредственное конструирование ключевых показателей эффективности (KPI), выполняющих функцию связующего звена между техническими SLO и высокоуровневыми бизнес-метриками. Именно на этом этапе абстрактное «сократить время загрузки» трансформируется в конкретное, экономически обоснованное «снизить показатель отказов мобильной аудитории на пять процентных пунктов за счёт удержания P90 LCP ниже двух с половиной секунд». Например, KPI может звучать так: «Сокращение показателя отказов (bounce rate) на мобильных устройствах на 5% за счёт удержания P90 LCP ниже 2.5 секунд». На Рисунке 2.3 показана иерархия целей, связывающая бизнес-уровень с операционным и техническим.

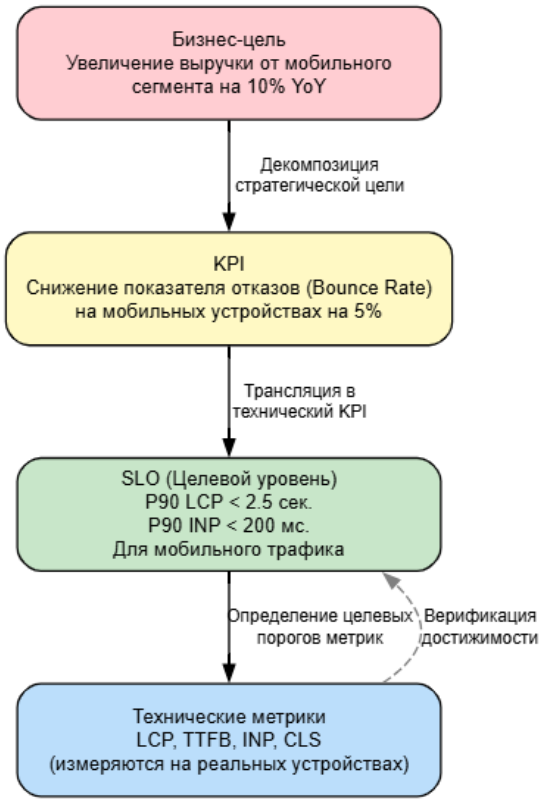

Рис.2.3. Иерархия целей: от бизнес-стратегии к техническим метрикам

§2.5. Математический фундамент: от физики распространения сигнала до закона Литтла

Проектирование высокопроизводительных веб-приложений требует выхода за рамки чисто эмпирических методов и обращения к фундаментальным законам, диктуемым физической природой вычислительных и коммуникационных процессов. Оптимизация, лишённая математического базиса, редуцируется до набора несистематизированных практик, не гарантирующих устойчивого результата в условиях вариативной нагрузки. Данный раздел посвящён анализу неустранимых физических ограничений и формальных моделей, позволяющих прогнозировать поведение системы и верифицировать эффективность оптимизационных стратегий.

Физические пределы скорости распространения сигнала.

Ключевым и принципиально неустранимым фактором, лимитирующим скорость отклика распределённой системы, является конечная скорость распространения электромагнитных волн в среде передачи. Данное ограничение не может быть скомпенсировано исключительно программными методами и требует пространственной аппроксимации данных к пользователю. Время распространения сигнала (T_{prop}) в оптоволоконной линии связи определяется соотношением:

$$T_{prop} = \frac{D}{v} = \frac{D \cdot n}{c}$$

где D — физическая протяжённость канала, v — скорость света в волокне, c — скорость света в вакууме (3×10^8 м/с), а n ≈ 1.5 — показатель преломления сердцевины оптического волокна. Из этого следует, что пакет преодолевает расстояние в 1000 км по прямому оптическому кабелю не менее чем за 5 мс. В реальных топологиях с множественными узлами маршрутизации и коммутации эта задержка, называемая Round-Trip Time (RTT), мультипликативно возрастает. Данный фундаментальный предел обосновывает необходимость применения Content Delivery Networks (CDN) и географически распределённых архитектур, которые не столько «ускоряют» передачу, сколько минимизируют параметр D в приведённой формуле. Игнорирование данного ограничения при проектировании, например, размещение серверов для азиатского рынка в европейском дата-центре, делает невозможным достижение целевых показателей TTFB вне зависимости от качества серверной оптимизации.

Моделирование серверной обработки на основе теории массового обслуживания.

В то время как задержка распространения детерминирована расстоянием, задержка обработки запросов на серверной стороне носит стохастический характер, обусловленный конкурентным характером поступления заявок. Адекватным математическим аппаратом для анализа данного процесса служит теория массового обслуживания (ТМО). Простейшей, но чрезвычайно полезной для инженерных оценок моделью является система M/M/1, описывающая очередь с одним обслуживающим прибором, пуассоновским входящим потоком заявок с интенсивностью λ и экспоненциально распределённым временем обслуживания с интенсивностью μ.

Ключевым показателем, определяющим стабильность системы, является коэффициент загрузки ρ: $\rho = \frac{\lambda}{\mu}$. При приближении ρ к единице время пребывания заявки в системе (T_{sys}) нелинейно возрастает, что описывается формулой: $T_{sys} = \frac{\frac{1}{\mu}}{1-\rho}$. Из данной зависимости следует критически важный для практической оптимизации вывод: при увеличении загрузки сервера с 50% до 90% среднее время отклика возрастает не в 1.8, а в 5 раз. Это объясняет эффект резкой, «обвальной» деградации производительности в моменты пиковых нагрузок и обосновывает необходимость поддержания эксплуатационно-

го запаса по мощности, а также применения механизмов ограничения входящего потока (rate limiting, circuit breaker).

Фундаментальное соотношение, связывающее среднее число заявок, единовременно пребывающих в системе (L), со средним временем их персистенции в ней (W), формализуется законом Литтла, который обладает статусом математического инварианта, справедливого для произвольных стационарных систем массового обслуживания вне зависимости от их внутренней архитектуры, дисциплины диспетчеризации очереди или конкретного вида вероятностных распределений входящего потока и времени обслуживания: $L = \lambda \cdot W$. Данный закон предоставляет в распоряжение инженера по производительности исключительно мощный эвристический инструментарий, позволяющий осуществлять верификацию корректности и внутренней согласованности эмпирических измерений, а также строить валидные модели «чёрного ящика» для систем, внутренняя структура которых недоступна для прямого наблюдения. Так, располагая измеренными значениями среднего количества одновременных, конкурентных соединений с базой данных и среднего времени выполнения одного запроса, исследователь получает возможность вычислить фактическую, реальную пропускную способность подсистемы хранения, не прибегая к интрузивному анализу её внутренней архитектуры, планов выполнения запросов или конфигурации индексов. Таким образом, последовательное применение аналитического аппарата теории массового обслуживания обеспечивает методологическую основу для перехода от реактивной парадигмы — устранения уже случившихся сбоев и деградаций — к проактивному, предиктивному проектированию систем, характеризующихся наперёд заданными, формально верифицированными параметрами надёжности и производительности.

Графическая экспликация модели M/M/1.

На Рисунке 2.4 экспонируется ориентированный размеченный граф состояний для марковской системы массового обслуживания типа M/M/1. Данный граф наглядно иллюстрирует дискретные переходы системы между состояниями S_k, характеризуемыми наличием ровно k заявок (в очереди и на обслуживании), под совокупным воздействием двух конкурирующих стохастических процессов: пуассоновского входящего потока с интенсивностью λ (переходы $S_k \to S_{k+1}$) и экспоненциально распределённого потока обслуживания с интенсивностью μ (переходы $S_k \to S_{k-1}$).

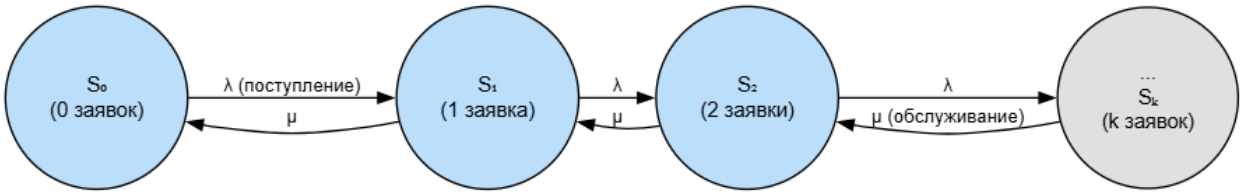

Рис. 2.4. Размеченный граф состояний системы массового обслуживания типа M/M/1

Модели сетевого взаимодействия и пропускная способность.

Ограничения пропускной способности канала также поддаются формальному анализу. Максимальный объём данных (M_{data}), который может быть передан за время T в канале с пропускной способностью B и RTT, в самом упрощённом виде ограничивается произведением полосы пропускания на задержку (Bandwidth-Delay Product, BDP):

$$BDP = B \times RTT$$

Величина BDP определяет объём данных, который должен находиться «в полёте», чтобы полностью утилизировать канал. Несоответствие размера TCP-окна перегрузки величине BDP является частой причиной неэффективного использования дорогостоящих высокоскоростных каналов с большими задержками (например, спутниковых). Оптимизация на данном уровне заключается в настройке алгоритмов управления перегрузкой (BBR, CUBIC) и буферов, что позволяет аппроксимировать реальную скорость передачи данных к теоретически достижимой границе, задаваемой теоремой Шеннона-Хартли для канала с аддитивным белым гауссовским шумом. Таким образом, математический фундамент является не абстрактным дополнением, а единственным легитимным инструментом для принятия обоснованных архитектурных решений, направленных на достижение предельно возможной производительности в условиях заданных физических ограничений.

ВЫВОДЫ

Вторая глава заложила метрологический фундамент дисциплины, без которого любые рассуждения о производительности остаются спекулятивными. Ключевым концептуальным достижением явилось введение расширенной таксономии метрик, не ограничивающейся дихотомией «лабораторное — полевое», а включающей синтетический мониторинг в продуктовом окружении как самостоятельный, комплементарный класс измерений. Было показано, что только синергия всех трёх подходов обеспечивает полноту кар-

тины: от детерминированной воспроизводимости лаборатории до репрезентативной, хотя и зашумлённой, картины реального пользовательского опыта.

Семантический анализ Core Web Vitals вывел рассмотрение за рамки технических дефиниций. Было установлено, что LCP является операционализацией понятия «перцептивная готовность», INP — непрерывной отзывчивости, а CLS — визуального доверия. Такая семантическая интерпретация позволяет инженеру соотносить численные значения метрик не с абстрактными порогами, а с качественными состояниями пользовательского восприятия, что радикально меняет приоритизацию оптимизационных задач.

Математический аппарат, представленный в параграфе 2.5, выполняет в структуре пособия функцию, далеко выходящую за рамки иллюстративной. Формулы расчёта времени распространения сигнала и времени пребывания заявки в системе M/M/1 являются не теоретическими артефактами, а практическими инструментами прогнозирования. Осознание неустранимости физических ограничений, диктуемых конечностью скорости света и нелинейностью функции $\frac{1}{1-\rho}$, формирует у обучающегося инженерную интуицию, позволяющую на этапе архитектурного проектирования отсеивать заведомо нереализуемые решения без затрат на их экспериментальную проверку. Закон Литтла, в свою очередь, предоставляет эвристику для верификации корректности измерений «чёрного ящика», что является незаменимым навыком при аудите систем, внутренняя архитектура которых недоступна.

Вопросы для самопроверки

1. Проведите сравнительный анализ лабораторных и полевых измерений производительности. В каких ситуациях каждый из подходов является предпочтительным?
2. Что понимается под «синтетическими измерениями в продуктовом окружении» (Synthetic Monitoring)? Какое место они занимают в таксономии метрик?
3. Раскройте семантику метрики Largest Contentful Paint (LCP). Почему она считается более репрезентативной для оценки пользовательского опыта, чем First Contentful Paint (FCP)?
4. В чём заключается фундаментальное ограничение метрики First Input Delay (FID) и как метрика Interaction to Next Paint (INP) его преодолевает?
5. Дайте определение метрике Cumulative Layout Shift (CLS). Объясните, как рассчитываются её компоненты: impact fraction и distance fraction.

6. Опишите иерархию целей от бизнес-стратегии к техническим метрикам. Что такое SLO и SLA, и в чём их принципиальное различие?
7. Почему использование среднего арифметического (average) для временных метрик категорически не рекомендуется? Какой статистический показатель следует использовать вместо него?
8. Выведите формулу для расчёта времени распространения сигнала в оптоволоконной линии связи. Какие физические константы в неё входят?
9. Опишите модель массового обслуживания M/M/1. При каких условиях она применима для анализа веб-серверов?
10. Сформулируйте закон Литтла и приведите пример его практического применения для верификации корректности измерений в веб-системе.

Задания на количественный анализ

Задача 2.1. Расчёт RTT по географическому расстоянию.

Сервер расположен в Москве, пользователь — во Владивостоке. Расстояние по прямой — 6400 км. Показатель преломления оптоволокна $n = 1.48$. Определите минимально возможное время RTT, пренебрегая задержками на маршрутизаторах.

Задача 2.2. Анализ модели M/M/1.

Веб-сервер обрабатывает запросы со средней интенсивностью $\lambda = 50$ заявок/с. Среднее время обслуживания одной заявки — 15 мс. Определите: а) интенсивность обслуживания μ; б) коэффициент загрузки ρ; в) среднее время пребывания заявки в системе T_{sys}; г) среднее число заявок в системе L.

Задача 2.3. Нелинейность времени отклика.

Для системы M/M/1 с $\mu = 100$ заявок/с постройте таблицу зависимости среднего времени отклика T_{sys} от интенсивности входного потока λ для значений $\lambda = \{10,30,50,70,80,90,95,99\}$ заявок/с. Прокомментируйте характер нелинейности.

Задача 2.4. Применение закона Литтла.

Система мониторинга показывает, что среднее количество одновременных соединений с базой данных составляет $L = 12$, а среднее время выполнения запроса $W = 45$ мс. Определите фактическую интенсивность потока запросов λ. Соответствует ли она номинальной нагрузке в 300 запросов/с?

Задача 2.5. Расчёт Bandwidth-Delay Product.

Канал связи имеет пропускную способность $B = 100$ Мбит/с и $RTT = 30$ мс. Определите: а) величину BDP в мегабитах и мегабайтах; б) минимальный размер TCP-окна, необходимый для полной утилизации канала.

Задача 2.6. Оценка влияния TTFB на LCP.

Измерения показали, что TTFB составляет 800 мс, время загрузки и рендеринга LCP-элемента (изображения) — 1200 мс. После переноса сервера ближе к пользователю TTFB сократился до 200 мс. На сколько процентов улучшится LCP при прочих равных условиях?

Задача 2.7. Расчёт CLS для единичного сдвига.

Внезапно загрузившийся рекламный баннер высотой 150 пикселей сдвинул весь контент вниз. Высота viewport составляет 900 пикселей. Баннер появился в верхней части экрана. Определите значение CLS для этого инцидента.

Задача 2.8. Анализ процентилей.

Дана выборка времён ответа сервера (в мс): 45, 48, 50, 52, 55, 58, 60, 62, 65, 70, 75, 80, 85, 90, 100, 120, 150, 200, 350, 1200. Определите: а) среднее арифметическое; б) медиану (P50); в) P75; г) P95; д) P99. Объясните, почему среднее арифметическое является плохим показателем для данной выборки.

Задача 2.9. Интерпретация SLO.

Команда установила SLO: «P95 LCP не должен превышать 3.0 секунды за 28-дневное скользящее окно». За месяц собраны данные: 100 000 измерений LCP, из которых ровно 6000 превысили порог в 3.0 секунды. Находится ли система в рамках SLO? Обоснуйте ответ расчётом.

Задача 2.10. Расчёт эффективности сжатия.

Исходный размер JavaScript-файла — 480 КБ. После минификации размер сократился до 320 КБ. Затем применено сжатие Brotli (коэффициент сжатия 0.22 от минифицированного размера). Определите: а) итоговый размер файла, передаваемый по сети; б) общий коэффициент сжатия относительно исходного файла.

ГЛАВА 3. СТРАТЕГИИ ОПТИМИЗАЦИИ КЛИЕНТСКОЙ ПОДСИСТЕМЫ

Клиентская подсистема веб-приложения, функционирующая в гетерогенной среде браузеров и пользовательских устройств, является тем компонентом, с которым конечный пользователь взаимодействует непосредственно. Именно здесь технические метрики, такие как время загрузки скриптов или объём передаваемых данных, трансформируются в субъективное восприятие скорости, плавности и отзывчивости интерфейса. Оптимизация клиентской стороны представляет собой многоаспектную задачу, охватывающую управление критическим путём рендеринга, выбор стратегий доставки ресурсов, реализацию эффективных моделей кэширования и применение архитектурных паттернов, минимизирующих вычислительную нагрузку на основной поток выполнения. Настоящая глава систематизирует данные стратегии, рассматривая их как взаимообусловленные элементы единой инженерной дисциплины.

§3.1. Управление критическим путём рендеринга (Critical Rendering Path)

Критический путь рендеринга (Critical Rendering Path, CRP) представляет собой детерминированную последовательность шагов, выполняемых браузером для преобразования полученных от сервера ресурсов (HTML, CSS, JavaScript) в визуализированные пиксели на экране пользователя. Понимание этой последовательности и факторов, блокирующих её прохождение, является краеугольным камнем клиентской оптимизации. Любая задержка на любом из этапов CRP напрямую увеличивает время до первого контентного отображения (FCP) и, как следствие, ухудшает пользовательский опыт.

Процесс начинается с получения HTML-документа и его пошагового парсинга. Парсер HTML строит объектную модель документа (Document Object Model, DOM), представляющую собой древовидную структуру всех элементов страницы. Параллельно с этим, как только парсер обнаруживает ссылки на внешние таблицы стилей (<link rel="stylesheet">), инициируется их загрузка и построение объектной модели CSS (CSS Object Model, CSSOM). Принципиально важным для понимания механики браузерного рендеринга является то обстоятельство, что конструирование объектной модели CSS (CSSOM) представляет собой операцию, блокирующую последующую отрисовку: браузерный агент не может инициировать визуализацию контента на

экране до тех пор, пока построение CSSOM не будет полностью завершено. Данное ограничение носит фундаментальный, архитектурный характер и обусловлено тем, что результирующее дерево рендеринга (Render Tree), образующееся путём слияния двух древовидных структур — объектной модели документа (DOM) и объектной модели стилей (CSSOM), — обязано содержать финальные, вычисленные (computed) значения всех стилевых атрибутов для каждого визуально отображаемого элемента. Именно это фундаментальное ограничение эксплицирует, почему неоптимизированные, избыточные по объёму каскадные таблицы стилей, в особенности загружаемые из внешних, территориально удалённых источников, характеризующихся высокой сетевой латентностью, выступают в качестве одного из доминирующих факторов, лимитирующих скорость первой контентной отрисовки (First Contentful Paint).

Присутствие JavaScript вносит дополнительный, нетривиальный уровень сложности во взаимодействие компонентов критического пути рендеринга. В конфигурации по умолчанию синхронно загружаемые скрипты, специфицированные тегом <script src="..."> без модифицирующих атрибутов async или defer, классифицируются как блокирующие парсер (parser-blocking). При детектировании такого тега HTML-парсер принудительно приостанавливает свою работу до момента полной загрузки и последующего исполнения скрипта. Данное поведение продиктовано тем, что JavaScript потенциально способен осуществлять мутацию как DOM (например, посредством вызова document.write или манипуляций с узлами), так и CSSOM (через запросы вычисленных стилей элементов). Поскольку CSSOM, как было установлено выше, блокирует рендеринг, а JavaScript, в свою очередь, может находиться в зависимости от готовности CSSOM (например, при попытке программного доступа к вычисленным стилям элемента), в системе возникает феномен каскадной, или вторичной, блокировки: скрипт, расположенный в HTML-разметке после внешней таблицы стилей, не может начать своё исполнение до тех пор, пока построение CSSOM не будет полностью финализировано. Таким образом, CSS косвенно, опосредованно задерживает не только рендеринг, но и выполнение JavaScript-кода. Таким образом, CSS косвенно задерживает исполнение JavaScript.

Оптимизация CRP заключается в минимизации количества и размера критических ресурсов, необходимых для первоначальной отрисовки, и управлении порядком их загрузки. Ключевые техники включают:

- Инлайнинг критического CSS (Critical CSS): Выделение стилей, необходимых для отрисовки верхней части страницы (above the fold), и их встраивание непосредственно в HTML-документ в теге <style>. Остальные, некритические стили загружаются асинхронно с низким приоритетом.
- Асинхронная и отложенная загрузка скриптов: Использование атрибутов async (скрипт загружается параллельно с парсингом и выполняется немедленно после загрузки, не гарантируя порядок) и defer (скрипт загружается параллельно, но выполняется строго после завершения парсинга HTML и в порядке их следования). Это устраняет блокировку парсера.
- Предварительная загрузка критических ресурсов: Использование <link rel="preload"> для ранней инициации загрузки ресурсов, которые понадобятся в ближайшее время (например, шрифтов или изображений-заставок), что позволяет браузеру обнаружить их до того, как до них дойдёт парсер.

На Рисунке 3.1 представлена упрощённая диаграмма последовательности этапов критического пути рендеринга, иллюстрирующая взаимодействие между DOM, CSSOM и JavaScript.

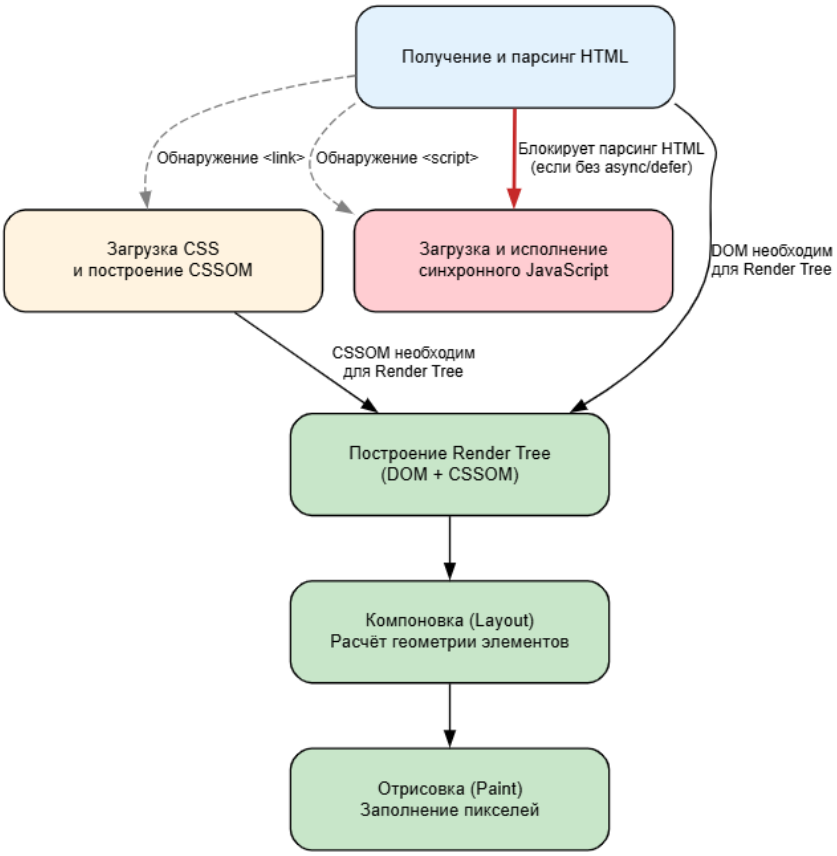

Рис.3.1. Упрощённая диаграмма критического пути рендеринга

§3.2. Эффективные стратегии доставки статических ресурсов: сжатие, форматы, CDN

После того как критический путь рендеринга оптимизирован с точки зрения порядка загрузки, следующим объектом внимания становится минимизация объёма передаваемых по сети данных. Статические ресурсы — изображения, скрипты, стили, шрифты — составляют подавляющую часть трафика современных веб-приложений, и их неэффективная доставка напрямую увеличивает метрики LCP и TTI.

Оптимизация изображений: форматы и адаптивность.

Изображения являются доминирующим по объёму типом контента на большинстве сайтов. Выбор формата кодирования изображения оказывает решающее влияние на баланс между визуальным качеством и размером файла. Традиционные форматы JPEG и PNG уступают место современным кодекам:

- WebP: Разработанный Google формат, обеспечивающий сжатие без потерь и с потерями. В среднем WebP-изображения на 25-35% меньше по размеру по сравнению с аналогами в JPEG и PNG при сопоставимом визуальном качестве. Поддерживает прозрачность (альфа-канал) и анимацию.
- AVIF (AV1 Image File Format): Формат следующего поколения, основанный на видео-кодеке AV1. Этот формат обеспечивает ещё более высокую степень компрессии, достигающую 50% от исходного объёма файла при сопоставимых показателях визуального качества. Кроме того, хоть и нативно, но поддерживает некоторые расширенные функции: включая режим динамического диапазона (High Dynamic Range, HDR) и широкий цветовой охват (Wide Color Gamut), что делает его особенно предпочтительным для устройств с дисплеями последнего поколения (например: Смартфоны Apple, начиная с моделей iPhone 12 (2020) и новее; Samsung Galaxy S и Galaxy Z Fold/Flip, начиная с серии Galaxy S21 (2021) и новее; Google Pixel, начиная с Pixel 6 / Pixel 6 Pro (2021) и вплоть до актуальных Pixel 8 Pro (2023) и Pixel 9 (2024), Ноутбуки и моноблоки, оснащённые дисплеями Pro Display XDR или Liquid Retina XDR).

Вместе с тем, выбор оптимального формата кодирования представляет собой лишь вершину айсберга в целостной стратегии по оптимизации. Не менее существенное значение приобретает адаптивность. Задействование

HTML-атрибутов *srcset* и семантического элемента <*picture*> делегирует браузеру полномочия по автоматическому, контекстно-зависимому выбору наиболее подходящего файла изображения из предварительно сгенерированного набора вариантов, основываясь на текущих геометрических размерах области просмотра (viewport) и пиксельной плотности (Device Pixel Ratio, DPR) клиентского устройства. Данный механизм эффективно предотвращает загрузку полноформатных, многомегабайтных изображений на мобильные терминалы, оснащённые экранами с небольшой физической диагональю и, зачастую, лимитированными сетевыми каналами. Генерация требуемой номенклатуры вариантов изображения — различных разрешений, форматов и степеней сжатия — должна быть в обязательном порядке автоматизирована и интегрирована либо в конвейер этапа сборки (build-time генерация), либо реализована посредством специализированных сервисов динамической обработки изображений класса Image CDN (таких как Cloudinary или imgix), которые способны осуществлять трансформацию размера, формата и уровня компрессии «на лету», параметризуясь исключительно строкой запроса URL.

Минификация и сжатие текстовых ресурсов.

Скрипты и стили, прежде чем попасть к пользователю, должны пройти этап минификации — процесс удаления всех синтаксически незначимых символов (пробелов, переносов строк, комментариев) без изменения функциональности кода. Инструменты, такие как Terser (для JavaScript) и cssnano (для CSS), интегрируются в сборочные конвейеры (Webpack, Vite) и обеспечивают сокращение размера файлов на 20-50%. Дополнительно применяется транспортное сжатие на уровне HTTP. Алгоритмы Gzip и Brotli (последний демонстрирует на 15-25% лучшую степень сжатия для текстовых данных) настраиваются на стороне веб-сервера или CDN.

Роль CDN в доставке контента.

Content Delivery Network (CDN) выступает как географически распределённый кэш, а также как полноценный слой оптимизации доставки. Современные CDN-провайдеры (Cloudflare, Akamai, Fastly и др.) выполняют на граничных узлах ряд критически важных функций: создают и терминируют TLS-соединения «рядом» с пользователем, тем самым сокращая задержки рукопожатия; автоматически применяют наиболее эффективный алгоритм сжатия (например использованием технологии Brotli); конвертируют изображения в современные форматы «на лету»; обеспечивают защиту от DDoS-атак. Поэтому, можно сказать, что CDN минимизирует не только физическую дистанцию (RTT), но и объём передаваемых данных, а также снижает

нагрузку на исходный сервер, забирая на себя обработку подавляющей доли статических запросов.

§3.3. Архитектурные шаблоны кэширования на стороне клиента

Кэширование на стороне клиента является одним из наиболее эффективных методов сокращения количества сетевых запросов и ускорения загрузки при повторных посещениях. В отличие от серверного кэширования, оперирующего агрегированными данными для множества пользователей, клиентский кэш управляется индивидуально каждым браузером на основе директив, полученных от сервера. Грамотно спроектированная стратегия кэширования позволяет полностью исключить передачу неизменившихся ресурсов, сокращая не только трафик, но и время, затрачиваемое на сетевые взаимодействия.

Основным механизмом управления является HTTP-заголовок Cache-Control. Директива max-age=<seconds> предписывает браузеру считать ресурс «свежим» в течение указанного количества секунд с момента его получения. В течение этого периода браузер обслуживает запросы к данному ресурсу из локального кэша, не обращаясь к серверу вовсе. Эта стратегия, известная как «долгосрочное кэширование» (Long-lived Caching), является оптимальной для статических ресурсов с детерминированными именами. Для её реализации применяется техника хеширования имён файлов: в имя файла бандла (например, app.a1b2c3d4.js) включается хеш его содержимого. При любом изменении кода хеш изменится, URL ресурса станет другим, и браузер загрузит новую версию, в то время как старые версии могут кэшироваться с очень большим max-age (вплоть до года). Это помогает разрешить противоречие между актуальностью и производительностью.

Для ресурсов, которые могут изменяться недетерминировано т.е. непредсказуемо и для которых критична актуальность (например, ответы API интерфейса), применяется стратегия валидации (Conditional Requests). Сервер вместе с ответом передаёт заголовок ETag (уникальный идентификатор версии ресурса, часто хеш) или Last-Modified (временная метка последнего изменения). Затем, при последующем запросе, браузер отправляет условные заголовки If-None-Match (со значением сохранённого ETag) или If-Modified-Since, если ресурс не изменился, сервер отвечает статусом 304 Not Modified с пустым телом, что значительно быстрее полной передачи данных. Эта стратегия, именуемая как «stale-while-revalidate» (поддерживаемом через Cache-Control: stale-while-revalidate=<seconds>), позволяет браузеру мгновенно от-

давать пользователю кэшированную версию, одновременно асинхронно проверяя наличие обновлений в фоновом режиме.

Подобный подход демонстрирует эффективность приложения к тем категориям данных, для которых спецификацией предметной области допускается кратковременная задержка между состояниями. К числу таких категорий относятся, в частности, ленты новостных агрегаторов, списки рекомендуемых товаров в электронной коммерции, ленты публикаций в социальных платформах и иные сценарии, где безусловный примат скорости доставки контента над его абсолютной, сиюминутной актуальностью является осознанным архитектурным компромиссом.

Обобщённая схема, эксплицирующая взаимосвязи между различными стратегиями клиентского кэширования и их релевантность к тому или иному типу ресурсов, представлена в графической форме на Рисунке 3.2.

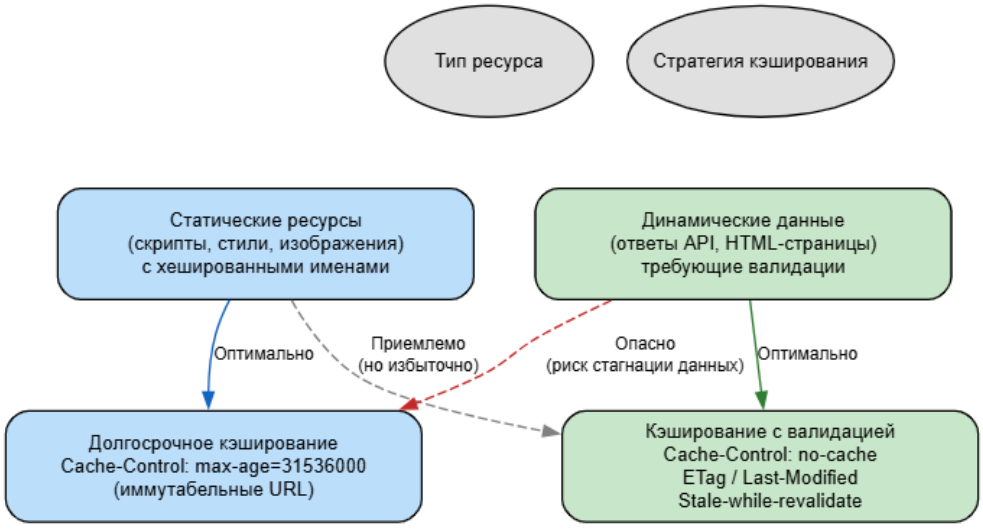

Рис.3.2. Матрица принятия решений при выборе оптимальной стратегии клиентского кэширования в зависимости от типа ресурса и требований к актуальности данных.

§3.4. Декомпозиция программного кода: отложенная загрузка, динамический импорт и элиминация мёртвого кода

Объём JavaScript-бандла, доставляемого пользовательскому агенту в процессе инициальной загрузки веб-страницы, выступает в качестве одной из критических, лимитирующих переменных, детерминирующих время достижения полной интерактивности интерфейса (Time to Interactive, TTI). В архитектуре крупных одностраничных приложений (Single Page Applications, SPA) монолитный бандл, агрегирующий всю кодовую базу в единый артефакт, способен достигать объёмов, исчисляемых мегабайтами, что закономерно индуцирует пролонгированные фазы лексического парсинга, синтак-

сического анализа и JIT-компиляции скриптов даже после формального завершения их сетевой передачи. Декомпозиция кода представляет собой семейство методологически связанных техник, объединённых единой целевой установкой: разбиением исходно монолитной кодовой базы на множество логически автономных, семантически завершённых фрагментов, загружаемых не единовременно, а по факту возникновения актуальной потребности в них — т.е. по требованию (on-demand).

Разделение кода (Code Splitting) и динамический импорт.

Техника разделения кода (Code Splitting) предоставляет разработчику возможность эксплицитно специфицировать в кодовой базе так называемые «точки разрыва» (split points), руководствуясь которыми, инструментарий этапа сборки (бандлеры Webpack, Rollup, Vite) осуществляет генерацию дискретных файлов-чанков (chunks). Стандартизированным и рекомендованным к повсеместному применению способом декларации подобных точек выступает синтаксис динамического импорта import('module-path'). В отличие от статического, директивного варианта импорта, размещаемого в прологе модуля, динамический импорт возвращает объект типа Promise, асинхронно разрешающийся целевым модулем.

Ленивая загрузка некритических ресурсов.

Помимо кода, ленивая загрузка активно применяется к изображениям, iframe и другим тяжёлым элементам, находящимся за пределами начальной области просмотра (below the fold). Нативный атрибут loading="lazy" для тегов <img> и <iframe>, поддерживаемый всеми современными браузерами, делегирует управление загрузкой самому браузеру, который использует внутренние эвристики для определения момента инициации запроса. Для более сложных сценариев, таких как загрузка компонента по видимости, используется Intersection Observer API, позволяющий эффективно отслеживать факт попадания элемента в viewport без ресурсоёмких вычислений на скролл-событиях.

Tree Shaking: элиминация мёртвого кода.

Дополняющей техникой, уменьшающей размер бандла не за счёт отложенной загрузки, а за счёт удаления неиспользуемого кода, является Tree Shaking. Этот термин, популяризированный экосистемой Webpack, описывает процесс статического анализа графа импортов модулей ES2015 с целью исключения из финального бандла тех экспортированных функций, классов

или переменных, которые нигде не импортируются. Ключевым условием эффективности Tree Shaking является использование статической структуры ES-модулей (import/export), которая может быть проанализирована на этапе сборки без исполнения кода. Модульные системы CommonJS (require) являются динамическими по своей природе и не поддаются надёжному статическому анализу, что делает их непригодными для tree shaking.

Разработчик должен осознанно проектировать модули так, чтобы они экспортировали «атомарные» функции, избегая сайд-эффектов на уровне модуля, которые могут быть ошибочно интерпретированы бандлером как необходимые и, следовательно, исключены из процесса удаления. Сочетание code splitting для разделения кода во времени и tree shaking для удаления неиспользуемого кода в пространстве формирует мощный тандем, направленный на минимизацию объёма кода, обрабатываемого браузером при старте приложения.

§3.5. Управление состоянием и мемоизация как методы снижения вычислительной сложности

Даже после того как код загружен, его неэффективное исполнение может стать причиной «подтормаживаний» интерфейса, проявляющихся в задержках реакции на пользовательский ввод (метрика INP) и низкой частоте кадров при анимациях и скролле. Управление состоянием и техники мемоизации направлены на минимизацию объёма повторных вычислений, выполняемых в ответ на изменения данных.

Проблема избыточных ререндеров в компонентных фреймворках.

В реактивных компонентных фреймворках (React, Vue, Svelte) изменение состояния компонента или его пропсов (входных параметров) запускает процесс повторного вычисления виртуального представления (Virtual DOM) и его сверки (reconciliation) с предыдущим состоянием. В ситуации, когда означенный процесс инициируется в отсутствие действительной, семантически обоснованной необходимости — например, в случае, если новое значение пропса, будучи ссылочно отличным от предыдущего (новая аллокация объекта или функции), тем не менее является семантически тождественным, — возникают избыточные, паразитные циклы рендеринга. Подобные циклы неоправданно потребляют дефицитные ресурсы центрального процессора клиентского устройства и способны продуцировать микрофризы, визуально воспринимаемые пользователем как «залипания» и прерывистость интерфейсных анимаций.

Мемоизация как инструмент когнитивной разгрузки вычислительного ядра.

Мемоизация представляет собой технику императивного кэширования результатов выполнения вычислительно ёмких, ресурсо-затратных функций с жёсткой ассоциативной привязкой сохранённого результата к конкретному кортежу входных аргументов, его породившему. При последующем обращении к функции с тождественным (либо эквивалентным в смысле поверхностного сравнения) кортежем входных параметров повторное вычисление не инициируется — вместо этого вызывающей стороне незамедлительно возвращается предварительно закэшированный, сохранённый в памяти результат. В контексте задач повышения производительности клиентской подсистемы означенный приём находит своё применение на нескольких комплементарных, взаимно усиливающих друг друга уровнях архитектурной абстракции:

Мемоизация на уровне компонентов: Обёрточные конструкции высшего порядка, а именно React.memo (для компонентов, воплощённых в функциональной парадигме) и PureComponent (для их классовых аналогов), реализуют автоматизированную процедуру поверхностного (shallow) сопоставления вновь поступившего набора пропсов с предшествующим. При детектировании их семантической тождественности компонент исключается из очереди на повторный цикл согласования виртуального DOM и, как следствие, ререндеринг. Описываемая техника демонстрирует максимальную прагматическую отдачу в приложении к компонентам, выполняющим роль акцепторов сложных, многосоставных, структурно разветвлённых, однако редко подвергающихся мутациям информационных объектов.

Мемоизация на уровне вычислений: Хук useMemo предоставляет механизм персистентного сохранения результата выполнения произвольной, потенциально ресурсоёмкой вычислительной процедуры — типичными примерами таковой могут служить операции фильтрации, сортировки или агрегации над массивами данных значительной размерности. Повторное исполнение означенной процедуры инициируется не на каждом цикле рендеринга, а исключительно при верифицированном изменении значений, специфицированных в массиве зависимостей данного хука. В периоды стабильности зависимостей результат извлекается из кэша, минуя фазу повторного счёта. Это предотвращает выполнение дорогостоящей логики на каждом рендере.

Мемоизация колбэков: Хук useCallback возвращает мемоизированную версию функции-обработчика. Это критически важно при передаче колбэк-

ов в дочерние компоненты, обёрнутые в React.memo: без useCallback новая функция будет создаваться на каждом рендере родителя, что приведёт к новому ссылочному значению пропса и инвалидации мемоизации дочернего компонента.

Селекторы и нормализация состояния.

На уровне архитектуры управления состоянием (Redux, Zustand, MobX) ключевым паттерном является использование селекторов — функций, которые извлекают и, при необходимости, вычисляют производные данные из глобального хранилища (store). Библиотека Reselect позволяет создавать мемоизированные селекторы, которые пересчитывают результат только при изменении соответствующих частей состояния. Это позволяет компонентам подписываться не на всё хранилище, а на минимально необходимый срез данных, избегая ненужных обновлений.

Дополнительным архитектурным решением, снижающим вычислительную сложность, является нормализация состояния. Вместо хранения данных в виде вложенных структур, где одни и те же сущности могут дублироваться в разных местах, применяется плоская структура, подобная реляционной базе данных: сущности хранятся в словарях, ключами которых являются их идентификаторы, а связи между ними описываются массивами ID. Подобная архитектурная организация радикально упрощает логику мутирования данных: модификация отдельно взятой сущности более не требует рекурсивного обхода и каскадного обновления всех потенциально затронутых, иерархически вложенных объектов, что, в свою очередь, кратно снижает вероятность возникновения лавинообразных, каскадных циклов ререндеринга, способных парализовать основной поток выполнения.

Синергетическое сочетание тщательно продуманной, нормализованной архитектуры глобального состояния с гранулярным, хирургически точным применением техник мемоизации — на уровнях компонентов, вычислений и колбэков — обеспечивает поддержание стабильно высокой частоты смены кадров (frame rate) и минимизацию латентности отклика на пользовательские взаимодействия. Достижение данных характеристик, в свою очередь, является необходимым и достаточным условием для соблюдения целевых пороговых значений метрики Interaction to Next Paint (INP) и формирования у конечного пользователя субъективного, трудно формализуемого, но критически значимого ощущения «бесшовности», непрерывности и монолитности интерфейса.

ВЫВОДЫ

Третья глава, посвящённая проблематике оптимизации клиентской подсистемы, эксплицировала фундаментальный тезис: результирующая эффективность стратегий ускорения на стороне браузерного агента детерминируется не механистическим, комбинаторным применением разрозненных техник, а глубинным, концептуальным пониманием внутренних механизмов функционирования платформы. Детальный анализ критического пути рендеринга вскрыл наличие фундаментальной асимметрии во взаимодействии его ключевых компонентов: каскадные таблицы стилей (CSS) блокируют непосредственно рендеринг, в то время как JavaScript блокирует синтаксический анализатор HTML (парсер), а их совместное, взаимно-обусловленное присутствие в критическом пути порождает феномен каскадных, вторичных блокировок, элиминация которых требует не косметических, точечных правок, а продуманных, системных архитектурных интервенций — таких как инлайнинг критических стилей в тело документа, принудительная асинхронная либо отложенная загрузка скриптов и упреждающая, спекулятивная загрузка критических субресурсов.

Рассмотрение стратегий доставки статических ресурсов выявило экономическую природу оптимизационных решений. Выбор между JPEG, WebP и AVIF, между Gzip и Brotli, между монолитным бандлом и гранулярными чанками — это всегда компромисс между степенью сжатия, вычислительными затратами на кодирование/декодирование и охватом поддерживающих платформ. Было показано, что CDN в современной архитектуре перестаёт быть пассивным кэшем, эволюционируя в активный слой, выполняющий трансформацию контента «на лету».

Завершающие разделы главы, посвящённые декомпозиции кода и управлению состоянием, резюмировали переход от парадигмы «оптимизации загрузки» к парадигме «оптимизации исполнения». Техники Code Splitting и Tree Shaking решают задачу минимизации объёма передаваемого и парсимого кода, в то время как мемоизация и нормализация состояния минимизируют объём повторно вычисляемого. Синтез этих двух направлений формирует целостный подход к клиентской производительности, где время до интерактивности (TTI) и задержка ответа на взаимодействие (INP) улучшаются не как побочный эффект, а как прямой результат осознанных архитектурных решений.

Вопросы для самопроверки

1. Дайте определение критического пути рендеринга (Critical Rendering Path). Перечислите его основные этапы в порядке их выполнения.
2. Объясните, почему построение CSSOM является блокирующей рендеринг операцией. Как это фундаментальное ограничение влияет на стратегию загрузки стилей?
3. В чём различие между атрибутами async и defer для тега <script>? В каких случаях следует применять каждый из них?
4. Опишите технику инлайнинга критического CSS (Critical CSS). Каковы её преимущества и потенциальные недостатки?
5. Проведите сравнительный анализ форматов изображений WebP и AVIF с точки зрения степени сжатия, визуального качества и поддержки браузерами.
6. Раскройте сущность стратегии «долгосрочного кэширования» (Long-lived Caching) с хешированием имён файлов. Почему она разрешает противоречие между актуальностью и производительностью?
7. Что такое Code Splitting и как динамический импорт (import()) реализует эту технику на практике?
8. Объясните принцип работы Tree Shaking. Почему данная техника эффективна только при использовании статической структуры ES-модулей?
9. Опишите проблему избыточных ререндеров в компонентных фреймворках. Какие инструменты предоставляет React для её решения?
10. В чём заключается архитектурный паттерн нормализации состояния в контексте управления данными на клиенте? Как он способствует снижению вычислительной сложности?

Задания на количественный анализ

Задача 3.1. Оценка влияния Critical CSS.

Общий размер CSS-файла — 180 КБ, из которых 25 КБ составляют критические стили (above the fold). Время загрузки полного файла — 400 мс. Если критические стили заинлайнены в HTML, а остальные загружаются асинхронно, на сколько миллисекунд сократится время до начала отрисовки (FCP)?

Задача 3.2. Оптимизация изображений.

Исходное JPEG-изображение имеет размер 850 КБ. Конвертация в WebP даёт сокращение размера на 30%, в AVIF — на 50% относительно

JPEG. Определите: а) размер файла в каждом формате; б) экономию трафика в мегабайтах на 10 000 показов для каждого формата относительно JPEG.

Задача 3.3. Расчёт эффективности адаптивных изображений.

Для десктопа изображение загружается в разрешении 1920×1080 (размер 450 КБ), для планшета — 1024×768 (210 КБ), для мобильного — 640×480 (85 КБ). Распределение трафика: десктоп — 40%, планшет — 25%, мобильные — 35%. Определите средний размер загружаемого изображения при использовании адаптивной стратегии и без неё (всегда десктопное).

Задача 3.4. Оценка выигрыша от сжатия Brotli.

Исходный размер JavaScript-бандла — 1.2 МБ. Коэффициент сжатия Gzip — 0.3, Brotli — 0.22. Определите: а) размер файла после каждого вида сжатия; б) на сколько процентов Brotli эффективнее Gzip для данного файла.

Задача 3.5. Расчёт hit-ratio кэша.

За период наблюдения кэш статических ресурсов получил 45 000 запросов, из которых 36 000 были обслужены из кэша (статус 200 from cache), 4 500 — с валидацией (статус 304 Not Modified), и 4 500 — полные загрузки (статус 200). Определите: а) hit-ratio (долю запросов, обслуженных без обращения к серверу); б) долю запросов, потребовавших передачи полного тела ответа.

Задача 3.6. Анализ эффективности Code Splitting.

Исходный бандл имел размер 2.0 МБ. После разделения на три маршрутных чанка размеры составили: main — 350 КБ, route-home — 200 КБ, route-dashboard — 800 КБ, route-settings — 650 КБ. Пользователи посещают: только home — 60%, home + dashboard — 25%, home + settings — 15%. Определите средний объём загружаемого кода на одну сессию.

Задача 3.7. Оценка влияния ленивой загрузки изображений.

Страница содержит 40 изображений за пределами начального viewport, каждое размером 120 КБ. Без ленивой загрузки все они загружаются сразу. С ленивой загрузкой в среднем загружается 8 из них (остальные не попадают в область видимости). Определите экономию трафика на одну сессию в мегабайтах.

Задача 3.8. Расчёт выигрыша от мемоизации.

Компонент без мемоизации ререндерится 120 раз в секунду, каждый рендер занимает 3.5 мс процессорного времени. После применения React.memo количество ререндеров сократилось до 15 в секунду. Определите: а) загрузку CPU (мс/с) до и после оптимизации; б) относительное снижение загрузки CPU.

Задача 3.9. Оценка эффекта виртуализации списка.

Список из 10 000 элементов отображается в контейнере, вмещающем 15 видимых элементов. Без виртуализации рендерятся все 10 000 DOM-узлов, время рендеринга — 850 мс. С виртуализацией рендерятся только 15 видимых узлов, время рендеринга пропорционально количеству узлов. Определите ожидаемое время рендеринга с виртуализацией.

Задача 3.10. Анализ влияния шрифтов на производительность.

Веб-шрифт загружается асинхронно, его размер — 85 КБ. Время загрузки — 600 мс. Во время загрузки текст не отображается (FOIT). После оптимизации применяется стратегия font-display: swap, и текст сразу отображается системным шрифтом, а через 600 мс заменяется веб-шрифтом. На сколько миллисекунд сократилось время до появления читаемого текста?

ГЛАВА 4. АРХИТЕКТУРНЫЕ АСПЕКТЫ СЕРВЕРНОЙ ОПТИМИЗАЦИИ

Если клиентская оптимизация, рассмотренная в Главе 3, направлена на сокращение времени восприятия и отзывчивости интерфейса, то серверная оптимизация решает фундаментальную задачу обеспечения стабильной, предсказуемой и масштабируемой обработки запросов в условиях варьирующейся нагрузки. Серверная подсистема, включающая уровень бизнес-логики, слой доступа к данным и сетевую инфраструктуру, является тем фундаментом, на котором базируется пользовательский опыт. Дефекты в её архитектуре, неоптимальные запросы к базе данных или отсутствие стратегии масштабирования неизбежно приводят к деградации метрик TTFB, увеличению частоты ошибок и, в конечном счёте, к финансовым потерям. Настоящая глава систематизирует ключевые архитектурные паттерны и инженерные практики, направленные на достижение высокой производительности и отказоустойчивости серверной стороны.

§4.1. Реляционные модели и физика выполнения запросов: индексы, планы, денормализация

Подавляющее большинство веб-приложений используют реляционные системы управления базами данных (СУБД) в качестве постоянного хранилища. Производительность операций чтения и записи в СУБД часто становится доминирующим фактором в общем времени отклика сервера. Понимание физических механизмов выполнения запросов, а не только их логической структуры на языке SQL, является обязательной компетенцией инженера по производительности.

Анатомия плана выполнения запроса.

Когда СУБД получает SQL-запрос, она не выполняет его непосредственно. Сначала планировщик запросов (Query Planner) генерирует одно или несколько альтернативных планов выполнения (Execution Plans) — последовательностей низкоуровневых операций, таких как последовательное сканирование таблицы (Seq Scan), сканирование по индексу (Index Scan), хеш-соединения (Hash Join) или сортировки. Для каждого плана на основе собранной статистики о распределении данных в таблицах вычисляется его стоимость (Cost) — условная величина, отражающая ожидаемые затраты процессорного времени и операций ввода-вывода. Выбирается план с минимальной оценённой стоимостью.

Ключевым аналитическим инструментом в арсенале разработчика, осуществимого переход от гадательных предположений к доказательной диагностике, выступает директива EXPLAIN (либо её расширенный вариант EXPLAIN ANALYZE, который не ограничивается экспозицией планируемого плана, но и осуществляет реальное исполнение запроса, возвращая фактические, инструментально зафиксированные темпоральные характеристики каждой элементарной операции). Скрупулёзный анализ плана выполнения позволяет дать исчерпывающие ответы на критически важные для оптимизации вопросы: задействуются ли существующие индексы, либо СУБД вынуждена прибегать к полному последовательному сканированию? Какой именно алгоритм соединения отношений (Join) — Nested Loop, Hash Join или Merge Join — был избран планировщиком? Каков кардинальный объём обрабатываемых данных на каждом дискретном шаге конвейера выполнения? Игнорирование данного инструментария и написание SQL-запросов «вслепую», основываясь исключительно на их логической корректности без учёта физической модели исполнения, является грубым, вопиющим нарушением фунда-

ментального принципа измеримости и не может быть квалифицировано иначе как профессиональная халатность.

Внутренняя механика и методология выбора индексов.

Под индексом в реляционной системе управления базами данных понимается специализированная, физически овеществлённая на дисковом либо оперативно-запоминающем носителе структура, чьё функциональное назначение заключается в обеспечении ускоренного, селективного доступа к кортежам целевой таблицы по предикатам, включающим значения одного или нескольких атрибутов. Данная структура позволяет полностью избежать выполнения чрезвычайно ресурсоёмкой операции полного последовательного обхода табличного пространства (Full Table Scan). В роли наиболее универсального и повсеместно применяемого типа индекса выступает B-tree — сбалансированное, сильно ветвящееся дерево поиска, демонстрирующее стабильно высокую эффективность в широком спектре операций: точечный поиск по ключу, выборка по диапазону значений и упорядочение результирующего набора. Принцип функционирования данной структуры зиждется на иерархической, многоярусной организации ключевых значений, где каждый узел дерева хранит отсортированный массив записей, содержащих указатели на дочерние узлы либо на физические локации кортежей. Ключевым свойством B-tree является то, что его глубина находится в строгой логарифмической зависимости от кардинальности индексируемой таблицы. Именно означенная логарифмическая асимптотика вычислительной сложности операций поиска и выступает гарантом сохранения стабильно высокой скорости доступа к данным даже в условиях экстремально больших объёмов последних, исчисляемых сотнями миллионов и миллиардами записей.

Выбор столбцов, которые будут включены в индекс, должен определяться не априорными предположениями разработчика, а напротив строгим количественным анализом профиля или для реальной эксплуатационной нагрузки. Композитные (составные) индексы, охватывающие несколько атрибутов, демонстрируют исключительную эффективность для запросов, осуществляющих фильтрацию по множественным полям в секции WHERE. Критически важным, однако, является порядок следования столбцов в таком индексе: он должен строго соответствовать порядку полей в предикатах запроса, начиная с наиболее селективных, то есть отсеивающих максимальный процент строк на ранних этапах. Покрывающие индексы (Covering Indexes) представляют собой дальнейшее развитие концепции: они включают в себя не только ключевые столбцы, но и все без исключения атрибуты, запрашива-

емые в секции SELECT. Это позволяет СУБД полностью удовлетворить запрос, оперируя исключительно данными, содержащимися в самой индексной структуре, без какого-либо обращения к основной таблице (операция Index Only Scan), что радикально, на порядки, сокращает количество дисковых операций ввода-вывода.

Вместе с тем, индексы не являются панацеей. Каждый дополнительно созданный индекс неизбежно замедляет выполнение операций модификации данных — вставки (INSERT), обновления (UPDATE) и удаления (DELETE), — поскольку СУБД вынуждена синхронно, в рамках той же транзакции, поддерживать консистентность всех индексных структур. Следовательно, стратегия индексации — это всегда поиск тонкого, динамического компромисса между скоростью операций чтения и скоростью операций записи, базирующийся исключительно на эмпирически измеренном профиле нагрузки конкретного приложения.

Нормализация и денормализация: диалектика проектирования схемы.

Нормализация данных, доведённая до третьей нормальной формы (3NF) либо, в более строгих случаях, до нормальной формы Бойса-Кодда (BCNF), является канонической, академически выверенной практикой проектирования реляционных схем, гарантирующей логическую целостность и минимизацию избыточности хранения. Однако высокий, педантичный уровень нормализации приводит к значительной фрагментации логически связанных данных по множеству физических таблиц, что для запросов, ориентированных на чтение или аналитическую обработку, оборачивается необходимостью выполнения многочисленных операций соединения (JOIN). Каждое такое соединение мультипликативно увеличивает вычислительную сложность запроса и при определённых условиях способно трансформироваться в узкое место.

В высоконагруженных системах с ярко выраженным доминированием операций чтения над операциями записи оправданным и рациональным является контролируемое, управляемое применение денормализации — преднамеренного, осознанного внесения контролируемой избыточности в схему данных с единственной целью ускорения читающих запросов. Это может выражаться в добавлении предвычисленных агрегирующих полей, дублировании данных из одной таблицы в другую или создании материализованных представлений (Materialized Views) — физически сохранённых на диске результатов выполнения дорогостоящих запросов. Платой за достигаемое ускорение чтения является усложнение логики записи, увеличение объёма храни-

мых данных и потенциальная угроза нарушения консистентности. Решение о применении денормализации должно приниматься исключительно на основе строгого количественного анализа: если конкретная операция JOIN является доказанным, инструментально подтверждённым узким местом, и её устранение даёт измеримый, статистически значимый прирост производительности, который перевешивает сопутствующие издержки по поддержанию избыточности.

§4.2. Иерархическое многоуровневое кэширование: от In-Memory Data Grids до CDN для динамического контента

Кэширование на серверной стороне представляет собой иерархически организованный, многоярусный процесс, телеологически направленный на обеспечение обслуживания максимально «горячих», наиболее часто запрашиваемых данных из наиболее быстрого и топологически близкого к точке потребления хранилища. Глобальная цель конструирования многоуровневой системы кэширования заключается в минимизации количества запросов, достигающих самого медленного, наиболее дорогого с точки зрения вычислительных и временны́х ресурсов слоя — дисковой подсистемы реляционной СУБД.

Уровень 1: Локальное кэширование в адресном пространстве приложения (In-Process Cache).

Простейший, начальный уровень кэширования реализуется непосредственно в оперативной памяти, в границах адресного пространства серверного процесса. Данные сохраняются в форме высокопроизводительных in-memory структур — хеш-таблиц или специализированных контейнеров, таких как LRU-кэш (Least Recently Used), автоматически вытесняющий наименее востребованные записи при исчерпании выделенного лимита памяти. Обращение к такому кэшу характеризуется латентностью, измеряемой наносекундами, что на несколько порядков превосходит по скорости любой сетевой запрос. Однако область применимости in-process кэша объективно ограничена: он не разделяется между множеством независимых экземпляров приложения и утрачивается при рестарте процесса. Данный уровень идеально подходит для хранения редко мутирующей, но часто запрашиваемой конфигурационной информации или эталонных справочных данных.

Уровень 2: Распределённые In-Memory Data Grids (Redis, Memcached).

Для обеспечения когерентности и консистентности кэша между множеством серверных инстансов и для персистентности данных между переза-

пусками приложений применяются выделенные, функционирующие как самостоятельные сетевые сервисы, in-memory хранилища. Redis, являясь дефакто отраслевым стандартом в данной категории, предоставляет не примитивное хранилище типа «ключ-значение», а богатый, выразительный набор высокоуровневых структур данных — списки, множества, сортированные множества, потоки — с поддержкой атомарных операций над ними. Кэширование результатов запросов к реляционной СУБД в Redis позволяет сократить нагрузку на последнюю на порядки. При проектировании такого кэша критически важным является выбор стратегии инвалидации. Наиболее надёжным и предсказуемым паттерном является Cache-Aside (Lazy Loading): приложение вначале проверяет наличие данных в кэше, и лишь в случае промаха загружает их из СУБД, после чего асинхронно либо синхронно сохраняет полученный результат в кэш. Инвалидация осуществляется явно при модификации данных в мастер-хранилище либо автоматически по истечении заданного времени жизни (TTL). Можно пойти иным путём — взять на вооружение схемы, где запись форсированно продавливается сквозь кэш напрямую в постоянное хранилище либо буферизуется для фоновой синхронизации. Подобные решения выправляют ситуацию с консистентностью, но расплачиваться за это приходится временем отклика на каждую операцию вставки или обновления, делая их ощутимо более медлительными.

Уровень 3: Кэширование на уровне HTTP и узлов CDN.

Выше, в Главе 3, уже шла речь о CDN как о разнесённом в пространстве накопителе для неизменяемых файлов. Но сегодняшние платформы доставки контента способны на большее — они вполне успешно справляются и с хранением динамически собираемых страниц. Выверенная установка управляющих заголовков позволяет периферийным серверам замыкать на себя обработку однотипных, не меняющихся от вызова к вызову запросов, разгружая центральный узел. Это даёт наибольший выигрыш на открытых API, где период годности данных жёстко задан. Помимо прочего, CDN берёт на себя роль демпфера при пиковых всплесках активности, срезая ударную волну и не пропуская её к серверам приложений. Объединение всех трёх звеньев — от внутрипроцессного хранилища до распределённой сети — порождает многоярусную, эшелонированную систему, призванную срезать время отклика до теоретического минимума для абсолютного большинства обращений.

§4.3. Горизонтальное масштабирование и балансировка

Свойство программной системы противостоять нарастающей нагрузке без зримых признаков деградации ключевых эксплуатационных характеристик образует фундаментальную детерминанту её долговременной жизнеспособности и итоговой экономической рациональности. Горизонтальное масштабирование (Scaling Out) — то есть экстенсивное приращение числа серверных узлов (инстансов), осуществляющих параллельную, конкурентную обработку входящего потока пользовательских заявок, — утвердилось в качестве отраслевого стандарта де-факто для подавляющего большинства веб-ориентированных систем. Указанный подход принципиально противопоставляется вертикальному масштабированию (Scaling Up), сводящемуся к монотонному увеличению вычислительной мощности и дисковой подсистемы единственной физической или виртуальной машины. Последнее неизбежно упирается в неустранимые физические лимиты — тепловыделение, потолок тактовых частот, пропускную способность шины памяти — и, что немаловажно, характеризуется экспоненциально возрастающей предельной стоимостью каждого дополнительного приращения ресурсов.

Балансировка нагрузки как диспетчер и регулятор трафика.

Балансировщик нагрузки (Load Balancer) материализует собой логически централизованную, унифицированную точку входа для всей совокупности клиентских запросов, осуществляя их прозрачную, бесшовную для конечного потребителя диспетчеризацию между пулом доступных, работоспособных серверных узлов. Избрание конкретной стратегии, или алгоритма, диспетчеризации оказывает подчас решающее воздействие на результирующую равномерность и, как следствие, эффективность утилизации агрегированных ресурсов всего вычислительного кластера.

Циклический обход (Round Robin): Наиболее примитивный, детерминированный до предела алгоритм, методично, по кругу, переадресующий каждый вновь прибывающий запрос следующему по списку серверу. Демонстрирует сносную, хотя и далеко не выдающуюся, эффективность в сугубо гомогенных средах — то есть там, где все запросы порождают примерно идентичную ресурсоёмкость, а все узлы кластера собраны по единому аппаратному лекалу.

Взвешивание по числу активных соединений (Least Connections): В отличие от слепого циклического перебора, данный алгоритм маршрутизирует входящий запрос тому узлу, который в настоящий момент времени удерживает наименьшее количество открытых, активных TCP-соединений. Такой

подход является существенно более интеллектуальным и адаптивным, поскольку имплицитно, без явного замера утилизации CPU, учитывает текущую фактическую загруженность каждого инстанса. Это качество приобретает критическую важность в сценариях, насыщенных долгоживущими, персистентными соединениями — такими, как WebSockets или Server-Sent Events (SSE).

Детерминированное хеширование (IP Hash / Consistent Hashing): Результат применения хеш-функции к IP-адресу клиента (либо к иному, семантически нагруженному ключу — скажем, токену или идентификатору сессии) служит базисом для строго детерминированного, воспроизводимого от запроса к запросу выбора целевого серверного узла. Данный метод нативно, без привлечения дополнительных внешних компонентов, реализует механизм так называемой «привязки сессии» (Session Stickiness) — свойства, являющегося безусловно необходимым для приложений, удерживающих локальное состояние (stateful) и не выносящих его во внешнее разделяемое хранилище. Алгоритм консистентного хеширования (Consistent Hashing), в отличие от своего упрощённого модульного собрата, решает фундаментальную проблему минимизации доли перераспределяемых ключей при изменении топологии кластера — добавлении нового узла или элиминации отказавшего. При простом модульном хешировании изменение общего числа узлов влечёт лавинообразную ремаппинг практически всех ключей; консистентное же хеширование ограничивает зону поражения лишь ближайшими соседями выбывшего или прибывшего узла. Данное свойство делает его архитектурно незаменимым, безальтернативным выбором для динамически масштабируемых, эластичных систем, топология которых перманентно флуктуирует.

Stateless - предпосылка к масштабированию

Краеугольным, не подлежащим ревизии архитектурным требованием, выполнение которого только и делает возможным по-настоящему эффективное горизонтальное масштабирование, выступает проектирование серверных компонентов в парадигме stateless — то есть лишённых какой бы то ни было формы локального, привязанного к конкретному инстансу хранения разделяемого состояния. Это означает, что каждый отдельный экземпляр приложения не должен хранить данные, необходимые для обслуживания последующих запросов от того же клиента. Любое состояние сессии должно быть вынесено во внешнее, разделяемое хранилище — распределённый кэш (Redis) или базу данных. При соблюдении этого принципа балансировщик может

направлять любой запрос от любого пользователя на любой свободный сервер, что обеспечивает максимальную гибкость и утилизацию.

В противовес этому, серверлесс-архитектура (Serverless), реализуемая в облачных платформах (AWS Lambda, Google Cloud Functions), доводит принцип stateless до логического предела. Код выполняется в виде короткоживущих, изолированных функций в ответ на события. Платформа автоматически управляет масштабированием, запуская столько экземпляров функции, сколько необходимо для обработки входящего потока. Это снимает с разработчика заботу об управлении серверами, но вносит проблему «холодного старта» (Cold Start) — задержки на инициализацию нового экземпляра функции, что требует специальных техник митигации (прогрев функций, использованиеProvisioned Concurrency).

§4.4. Проектирование API: REST vs GraphQL vs gRPC с позиции нагрузки

Выбор парадигмы проектирования API оказывает глубокое влияние на объём передаваемых данных, количество сетевых запросов и вычислительную нагрузку на сервер. Три доминирующих подхода — REST, GraphQL и gRPC — имеют принципиально разные компромиссы с точки зрения производительности.

REST (Representational State Transfer): предсказуемость и кэшируемость.

RESTful API, оперирующие ресурсами и использующие стандартные методы HTTP, являются наиболее распространённым архитектурным стилем. С точки зрения оптимизации, ключевыми преимуществами REST являются:

Идемпотентность и безопасность методов: Методы GET, PUT, DELETE являются идемпотентными, что позволяет прокси и клиентам безопасно повторять запросы при сбоях.

Кэширование на уровне HTTP: Ответы на GET-запросы могут эффективно кэшироваться на всех уровнях (браузер, CDN, прокси) с использованием стандартных заголовков Cache-Control, ETag.

Однако REST имеет и недостатки, главный из которых — избыточность (over-fetching) и недостаточность (under-fetching) данных. Фиксированная структура ответа часто содержит больше полей, чем необходимо клиенту. Для получения связанных данных может потребоваться несколько последовательных запросов (N+1 проблема на уровне API).

GraphQL: эффективность выборки ценой сложности сервера.

GraphQL решает проблемы over/under-fetching, позволяя клиенту в теле одного запроса специфицировать точную форму требуемых данных. Это минимизирует объём трафика и количество сетевых запросов. Однако эта гибкость переносит вычислительную сложность на сервер. Произвольно сконструированный GraphQL-запрос способен неявно спровоцировать каскад ресурсоёмких операций соединения (JOIN) и агрегации на стороне СУБД, сам факт возникновения и итоговую вычислительную стоимость которых крайне затруднительно ни предвидеть априорно, ни оптимизировать постфактум стандартными средствами планировщика запросов. Данное обстоятельство порождает нетривиальную проблему так называемых «дорогостоящих» запросов (Query Cost Analysis), для купирования которой требуется имплементация превентивных защитных механизмов. К числу последних относятся: принудительное лимитирование максимально допустимой глубины вложенности графа запроса, жёсткое квотирование кардинальности возвращаемых списковых структур и вычисление интегральной метрики «стоимости» запроса на этапе его статического анализа — до того, как он будет передан на исполнение движку базы данных. Дополнительным, зачастую упускаемым из виду ограничением выступает то обстоятельство, что GraphQL-запросы в своей канонической форме эксплуатируют HTTP-метод POST, что практически полностью нивелирует возможность их тривиального кэширования на промежуточных, транзитных узлах (прокси-серверах, CDN). Для восстановления утраченной кэшируемости приходится прибегать к специализированным архитектурным надстройкам — таким как персистированные запросы (Persisted Queries) или их автоматизированные аналоги (Automatic Persisted Queries, APQ), — которые, по существу, заменяют тело запроса его детерминированным идентификатором, вновь делая операцию безопасной и естественной для HTTP-семантики кэширования.

Фреймворк gRPC

gRPC представляет собой высокопроизводительный RPC-фреймворк, разработанный и поддерживаемый компанией Google, который опирается на два технологических столпа: механизм сериализации структурированных данных Protocol Buffers (protobuf) и протокол HTTP/2 в качестве транспортного уровня. Protobuf, в отличие от повсеместно распространённого текстового формата JSON, манипулирует бинарным, плотно упакованным представлением данных, что обеспечивает на порядки более высокую скорость как прямой сериализации, так и обратной десериализации, параллельно ра-

дикально сокращая физический объём передаваемой по сети полезной нагрузки. Задействование HTTP/2, в свою очередь, привносит в транспортный слой механизм мультиплексирования — возможность одновременной, конкурентной передачи множества независимых запросов и ответов в рамках единственного TCP-соединения, — что позволяет практически полностью элиминировать накладные расходы на установление новых соединений и связанные с ними задержки. Эти характеристики делают gRPC идеальным выбором для межсервисного взаимодействия в микросервисной архитектуре, где критичны пропускная способность и низкая латентность. Для взаимодействия с браузерами gRPC может быть менее удобен из-за необходимости использования gRPC-Web в качестве прокси-слоя.

Выбор между этими подходами должен диктоваться конкретным сценарием использования: REST — для публичных API, ориентированных на кэширование; GraphQL — для сложных клиентских интерфейсов с вариативными данными; gRPC — для внутреннего, высоконагруженного обмена между сервисами. Сравнительные характеристики подходов обобщены на Рисунке 4.1.

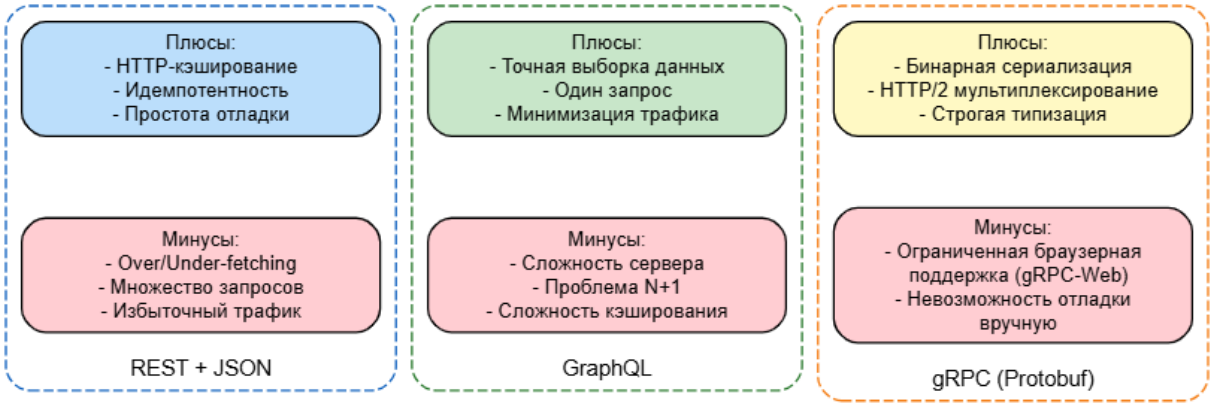

Рис.4.1. Сравнительный анализ парадигм проектирования API с позиции производительности

§4.5. Сетевые протоколы: эволюция от HTTP/1.1 к QUIC и HTTP/3, оптимизация TLS

Производительность веб-приложения неразрывно связана с эволюцией сетевых протоколов, которые определяют фундаментальные правила передачи данных. Понимание этой эволюции и осознанное использование возможностей современных протоколов является важной частью серверной оптимизации.

От HTTP/1.1 к HTTP/2: преодоление Head-of-Line Blocking.

Протокол HTTP/1.1, доминировавший в Вебе на протяжении многих лет, имел фундаментальное ограничение: в рамках одного TCP-соединения запросы обрабатывались строго последовательно. Задержка при обработке одного запроса блокировала все последующие (Head-of-Line Blocking, HOL Blocking). Обходным манёвром было открытие браузером нескольких параллельных TCP-соединений к одному хосту (обычно 6), что, однако, усложняло управление и не решало проблему полностью.

HTTP/2, стандартизированный в 2015 году, решил проблему HOL Blocking на прикладном уровне за счёт введения мультиплексирования потоков (Multiplexing). В рамках одного TCP-соединения данные множества запросов и ответов могут передаваться одновременно в виде небольших фреймов, чередующихся друг с другом. Это позволяет браузеру загружать все ресурсы параллельно через одно соединение, радикально повышая эффективность использования канала. HTTP/2 также вводит сжатие HTTP-заголовков (HPACK) и возможность принудительной приоритизации потоков сервером (Server Push), хотя последняя функция признана сложной в реализации и в значительной степени вытеснена <link rel="preload">.

QUIC и HTTP/3: преодоление TCP-обусловленного HOL Blocking.

Несмотря на все улучшения, HTTP/2 унаследовал проблему HOL Blocking на транспортном уровне, обусловленную устройством протокола TCP. TCP гарантирует строгую последовательность доставки байтов. Потеря одного пакета в потоке TCP приводит к тому, что все последующие пакеты, даже если они успешно доставлены, буферизуются до тех пор, пока потерянный пакет не будет передан повторно. Это означает, что потеря пакета, относящегося к одному ресурсу, способна заблокировать загрузку всех остальных ресурсов в том же соединении.

Протокол QUIC, разработанный Google и лёгший в основу стандарта HTTP/3, решает эту проблему радикально. QUIC работает поверх протокола UDP, а не TCP. Он реализует собственные механизмы надёжной доставки, но, в отличие от TCP, мультиплексирует независимые потоки данных уже на транспортном уровне. Потеря пакета в одном потоке QUIC не влияет на доставку данных в других потоках. Данное архитектурное решение полностью элиминирует феномен блокировки очереди запросов (Head-of-Line Blocking) на транспортном уровне — проблему, имманентно присущую протоколу TCP и не поддающуюся устранению в рамках предшествующих версий HTTP.

К числу сопутствующих, но от того не менее стратегически значимых преимуществ, индуцируемых внедрением связки QUIC/HTTP/3, правомерно отнести следующие.

Консолидация фазы установления соединения: QUIC осуществляет архитектурное слияние транспортного и криптографического (TLS 1.3) рукопожатий в одну атомарную, нерасчленимую процедуру. В наиболее благоприятном, хотя и не всегда достижимом на практике сценарии — при возобновлении соединения с сервером, с которым клиент уже взаимодействовал ранее и криптографические параметры которого сохранены в локальном кэше, — это позволяет выйти на режим 0-RTT (Zero Round-Trip Time). Суть последнего сводится к тому, что передача полезных, прикладных данных может быть инициирована немедленно, в самом первом отправляемом пакете, минуя какие-либо предварительные циклы обмена служебными сообщениями.

Устойчивость к миграции сетевого контекста (Connection Migration): В классической парадигме TCP соединение жёстко, нерасторжимо сцеплено с конкретной парой IP-адресов и номеров портов обеих взаимодействующих сторон. Любое изменение этого кортежа — например, вследствие переключения клиентского устройства из инфраструктуры Wi-Fi в сотовую сеть — неумолимо влечёт за собой разрыв соединения и необходимость его полного переустановления. QUIC разрывает эту жёсткую привязку, вводя абстрактный, независимый от сетевого уровня идентификатор соединения (Connection ID). Именно данное концептуальное новшество наделяет соединение свойством живучести при смене сетевого интерфейса: передача данных продолжается бесшовно, прозрачно для прикладного уровня, без потери контекста и без повторных рукопожатий. Для экосистемы мобильных приложений, где подобные переключения являются не исключением, а рутинной, повседневной нормой, означенная способность выступает критически значимым фактором, напрямую определяющим субъективное качество пользовательского опыта.

Оптимизация протокола TLS.

Обеспечение конфиденциальности и целостности соединения, гарантируемое протоколом Transport Layer Security (TLS), является неотъемлемым, императивным требованием к любой современной веб-системе. Вместе с тем, TLS неизбежно добавляет определённые накладные расходы, ассоциированные, прежде всего, с фазой криптографического рукопожатия. Минимизация этих издержек составляет самостоятельную задачу оптимизации.

Оптимизация TLS направлена на минимизацию этих издержек:

- TLS 1.3: Обязательное использование последней версии протокола, которая сокращает количество круговых обходов (round trips) при рукопожатии с двух до одного, а также удаляет устаревшие и небезопасные криптографические алгоритмы.
- TLS Session Resumption: Механизмы, позволяющие клиенту и серверу при повторном соединении использовать криптографические параметры, согласованные в ходе предыдущего рукопожатия, сокращая его до 1-RTT или даже 0-RTT (в сочетании с QUIC).
- OCSP Stapling: Вместо того чтобы браузер самостоятельно проверял статус отзыва сертификата, сервер периодически запрашивает OCSP-ответ у центра сертификации и «прикалывает» его к TLS-рукопожатию. Это экономит один сетевой запрос и сокращает задержку.

Эволюция протоколов от HTTP/1.1 к HTTP/3 и оптимизация TLS являются ярким примером того, как фундаментальные инженерные инновации на уровне инфраструктуры Интернета могут оказывать более значительное влияние на производительность, чем многие локальные оптимизации прикладного кода. Использование современных протоколов должно быть базовой, а не опциональной практикой.

§4.6. Использование Content Delivery Network (CDN) для ускорения доставки контента

Content Delivery Network представляет собой технологию географически распределенных серверов, оптимизирующую доставку контента конечным пользователям. Для ускорения доставки статических ресурсов задействуется архитектурное решение, известное как Content Delivery Network (CDN) — географически распределённая, иерархически организованная сеть серверных узлов, обеспечивающая быструю, с минимальной латентностью, доставку статичных данных конечным пользователям. Фундаментальный принцип функционирования CDN базируется на механизме упреждающей репликации статических артефактов — растровых изображений, каскадных таблиц стилей, исполняемых скриптов — на множество периферийных, граничных узлов (edge nodes), топологически приближенных к локациям концентрации целевой аудитории. Современные CDN-провайдеры существенно расширили изначально ограниченную функциональность, включив в неё полноценную поддержку обработки и доставки динамического, генерируемого сервером контента, что достигается посредством применения технологий

распределённого кэширования на граничных серверах, механизмов предварительного рендеринга страниц и алгоритмов интеллектуального, предиктивного прогнозирования пользовательских запросов. Подобная оптимизация позволяет мультипликативно снизить нагрузку на исходный, «материнский» сервер и радикально сократить время генерации и доставки контента для территориально удалённых пользователей.

Результирующая эффективность CDN находится в прямой, непосредственной зависимости от применяемых стратегий кэширования, охватывающих, в частности, тонкую настройку времени жизни кэшируемых объектов (TTL, Time to Live) и механизмов валидации контента. Оптимально подобранные параметры кэширования позволяют достичь искомого баланса между требованием актуальности доставляемых данных и императивом максимальной скорости их доставки. Для динамического контента, не допускающего агрессивного полного кэширования, применяются более изощрённые методы, такие как частичное кэширование фрагментов страниц (Edge Side Includes) или кэширование ответов на запросы с высокой степенью предсказуемости. Географическая распределённость узлов CDN минимизирует неизбежные сетевые задержки за счёт динамического выбора оптимального, с точки зрения текущей топологии и загруженности, маршрута доставки данных. Плотность размещения точек присутствия (Points of Presence, PoP) положительно коррелирует со степенью охвата целевой аудитории и, как следствие, со снижением ключевых метрик латентности. Современные системы маршрутизации CDN осуществляют автоматическое, основанное на мониторинге текущей сетевой конъюнктуры, направление пользовательских запросов к ближайшему доступному и наименее загруженному узлу.

§4.7. Балансировка нагрузки и масштабирование серверной инфраструктуры

Балансировка нагрузки выполняет функцию равномерного, сбалансированного распределения входящего потока пользовательских запросов между множеством серверных узлов, преследуя двуединую цель: повышение общей доступности сервиса и минимизацию времени отклика. В рамках оптимизации серверной подсистемы применяется спектр алгоритмов балансировки, включающий циклическое распределение (Round Robin), распределение с учётом текущего числа активных соединений (Least Connections) и распределение на основе вычисления хеш-функции от заданного ключа (Hash-based).

Корректная настройка таймаутов, алгоритмов распределения и мониторинга снижает риски перегрузок и обеспечивает предсказуемость работы сервиса.

Горизонтальное масштабирование предполагает наращивание числа инстансов приложения для распределения нагрузки и повышения отказоустойчивости. Вертикальное масштабирование заключается в увеличении вычислительных или дисковых ресурсов отдельного сервера и часто используется для быстрых приростов производительности без реархитектуры. Вертикальный подход ограничен физическими и экономическими факторами и может создавать единую точку отказа. Выбор между горизонтальным и вертикальным масштабированием определяется требованиями к доступности, сложностью управления и характером состояния приложений. Для систем со значительным состоянием применяются подходы с репликацией, шардированием или выделением специализированных сервисов, тогда как stateless-сервисы проще масштабировать горизонтально. Часто используется гибридная стратегия, сочетающая оба подхода и интегрирующая балансировщики, кэширование и распределение данных для оптимизации затрат и производительности.

«Бессерверная архитектура снижает эксплуатационные расходы, повышает производительность и позволяет разработчикам сосредоточиться на создании веб-приложений, не беспокоясь об инфраструктуре. Бессерверная парадигма выступает оптимальным архитектурным выбором для организаций, нацеленных на создание масштабируемых, экономически рациональных веб-приложений, эксплуатационные расходы на поддержку которых находятся в прямой, а не ступенчатой зависимости от фактической нагрузки. Автоматическое масштабирование в облачных средах конфигурируется на основе целевых, инструментально измеряемых метрик — степени утилизации процессора, объёма потребляемой памяти, латентности обработки запросов либо кастомных, бизнес-ориентированных индикаторов — и бесшовно интегрируется с балансировщиками нагрузки для динамического, упреждающего поддержания необходимого и достаточного пула вычислительных ресурсов. Подобный эластичный подход практически элиминирует необходимость в ручном, операторском вмешательстве и предоставляет инфраструктуре способность оперативно, в режиме, близком к реальному времени, адаптироваться к флуктуациям входящего трафика.

§4.8. Оптимизация API и протоколов взаимодействия

Выбор формата сериализации и передачи данных оказывает существенное, зачастую определяющее влияние на результирующую нагрузку серверной инфраструктуры. Архитектурный стиль RESTful API, несмотря на свою неоспоримую универсальность и широкую распространённость, нередко страдает от проблемы избыточности данных (over-fetching), порождаемой жёстко фиксированными, предопределёнными сервером структурами ответов. GraphQL предлагает принципиально иное решение данной проблемы, делегируя клиентскому приложению полномочия по спецификации точной формы и состава запрашиваемых данных, что позволяет минимизировать объём передаваемой по сети информации. Данное свойство приобретает особую актуальность в контексте мобильных приложений, функционирующих в условиях ограниченной пропускной способности и высокой латентности сотовых сетей. Оптимизация REST API, в свою очередь, достигается через внедрение механизмов пагинации, разреженной фильтрации полей (sparse fieldsets) и применение стандартных алгоритмов транспортного сжатия. Для GraphQL критически важным является внедрение ограничений на максимальную глубину вложенности запросов и реализация серверных механизмов кэширования для предотвращения генерации ресурсоёмких, многоуровневых вложенных запросов. Оба подхода императивно требуют тщательного, продуманного проектирования схемы данных для достижения оптимального баланса между гибкостью, выразительностью и производительностью. Только скрупулёзный анализ реальных, производственных сценариев использования позволяет осуществить обоснованный выбор оптимального протокола взаимодействия.

Сжатие ответов API с применением алгоритмов GZIP или Brotli обеспечивает сокращение объёма передаваемых данных на величину 60-80%, что непосредственно транслируется в уменьшение времени загрузки. Дополнительная минификация JSON-полезной нагрузки, заключающаяся в удалении синтаксически незначимых пробелов и сокращении имён ключей, позволяет осуществить дальнейшую оптимизацию размера ответа. При этом необходимо принимать во внимание вычислительные затраты на стороне сервера, ассоциированные с применением алгоритмов сжатия высокого уровня, и находить компромисс между степенью компрессии и затратами CPU. Корректная конфигурация HTTP-заголовков, в частности Content-Encoding, является обязательным условием для обеспечения прозрачной и корректной обработки сжатых данных всеми клиентскими агентами.

Асинхронная парадигма обработки запросов, реализуемая посредством внедрения специализированных очередей задач (таких как RabbitMQ или Apache Kafka), позволяет осуществить разгрузку основных серверных потоков, повышая тем самым общую отзывчивость API. Подобная организация вычислительного процесса открывает возможность для выполнения продолжительных, ресурсоёмких операций — таких как формирование объёмных аналитических отчётов, осуществление сложных многоэтапных вычислений или пакетная обработка значительных массивов растровых изображений — без какого-либо блокирующего воздействия на критический путь обслуживания пользовательских запросов. Задействование архитектурного паттерна «команда и запрос» (Command Query Responsibility Segregation, CQRS) проводит жёсткую, непреодолимую демаркационную линию между операциями, мутирующими состояние системы, и операциями, исключительно извлекающими данные, что создаёт предпосылки для независимой, селективной оптимизации каждого из этих классов. В свою очередь, агрегирование множества атомарных действий в рамках единой батчевой (пакетной) транзакции позволяет кратно, зачастую на порядок, сократить суммарное количество сетевых обращений. Например, массовая модификация записей через специализированные batch-эндпоинты, вместо последовательного вызова одиночных операций обновления, способна радикально, в разы, снизить пиковую нагрузку на подсистему персистентного хранения данных. Вместе с тем, данный подход требует тщательного, прецизионного контроля за размером пакетов во избежание таймаутов и перегрузки оперативной памяти сервера. Жёсткое лимитирование максимального количества операций в одном пакете является необходимым условием обеспечения стабильности системы.

§4.9. Сетевая оптимизация: протоколы, сжатие, мультиплексирование

Современные, эволюционно развитые протоколы передачи данных обеспечивают качественное, скачкообразное повышение эффективности сетевого взаимодействия. Протокол HTTP/2, пришедший на смену морально устаревшему HTTP/1.1, разрешил ключевые, фундаментальные ограничения предшественника за счёт внедрения механизма мультиплексирования множества независимых запросов в рамках единственного TCP-соединения. Данное нововведение позволяет полностью избежать проблемы блокировки очереди запросов (Head-of-Line Blocking) на прикладном уровне и осуществлять параллельную загрузку множества ресурсов. Помимо этого, HTTP/2 интродуцирует механизмы приоритизации потоков и компрессии служебных заголовков (HPACK), что дополнительно снижает задержки. Протокол QUIC, впоследствии стандартизированный как HTTP/3, предлагает следующий, ещё

более радикальный шаг в повышении производительности, функционируя поверх транспортного протокола UDP вместо TCP. Описанный архитектурный выбор обеспечивает минимизацию временны́х издержек на фазе постановки соединения за счёт слияния транспортного и криптографического (TLS) рукопожатий в одну слитную, атомарную процедуру, а также гарантирует встроенное, неотключаемое шифрование всего трафика и нативную, заложенную на уровне протокола резистентность к переключениям сетевого окружения — свойство, приобретающее критическую значимость именно в контексте мобильных платформ с их перманентной миграцией между точками доступа.

Задействование алгоритмов компрессии полезной нагрузки позволяет радикально, в разы, сократить физический объём пересылаемой информации, не поступаясь при этом ни единым битом её семантического наполнения. Такие техники, как GZIP и Brotli, демонстрируют впечатляющую результативность применительно к текстовым категориям ресурсов — HTML-документам, каскадным таблицам стилей и исполняемым скриптам. Brotli, в частности, выделяется на фоне предшественника аномально высокой степенью компрессии, превосходя GZIP на величину от одной пятой до одной четверти для статического, редко изменяемого контента. Безусловной, императивной нормой эксплуатации является конфигурирование веб-сервера на автоматическое, прозрачное для клиента применение сжатия ко всему спектру поддерживаемых MIME-типов. Синергия мультиплексирования потоков и транспортного сжатия позволяет достичь предельно возможной степени утилизации наличной пропускной способности канала, выжимая из него теоретический максимум пропускной способности.

В среде HTTP/2 мультиплексирование полностью устраняет необходимость в установлении множества параллельных TCP-соединений, снижая тем самым нагрузку на серверные ресурсы. Оптимальная конфигурация предполагает активацию сжатия заголовков HPACK и грамотную балансировку приоритетов потоков для обеспечения приоритетной доставки критических субресурсов. Эти меры совместно сокращают время загрузки страниц и экономят ресурсы сети.

ВЫВОДЫ

Четвёртая глава переместила фокус анализа на серверную подсистему, обосновав тезис о том, что архитектурные решения на уровне бэкенда имеют мультипликативный эффект, проявляющийся при масштабировании нагруз-

ки. Анализ физики выполнения SQL-запросов показал, что индексы не являются панацеей, а их выбор должен определяться статистическим профилем реальной рабочей нагрузки, а не априорными предположениями. Была эксплицирована диалектика нормализации и денормализации: первая минимизирует избыточность ценой усложнения запросов, вторая — ускоряет чтение ценой усложнения записи, и выбор между ними есть функция от преобладающего паттерна доступа к данным.

Рассмотрение иерархии кэширования — от наносекундного in-process до географически распределённого CDN — выявило эшелонированный характер защиты производительности. Каждый последующий уровень характеризуется на порядки большей ёмкостью, но и на порядки большей латентностью, что требует гранулярной настройки TTL и стратегий инвалидации. Было показано, что наиболее надёжным паттерном является Cache-Aside, оставляющий ответственность за консистентность на прикладном уровне.

Анализ алгоритмов балансировки нагрузки и условий эффективного горизонтального масштабирования привёл к фундаментальному выводу: stateless-архитектура является не опциональной практикой, а необходимым условием масштабируемости. Consistent Hashing был идентифицирован как алгоритм, минимизирующий цену реконфигурации кластера, что делает его предпочтительным для динамически масштабируемых систем.

Сравнительный анализ парадигм API и эволюции сетевых протоколов выявил, что технологический выбор на этих уровнях задаёт верхнюю границу достижимой производительности. Переход от HTTP/1.1 к HTTP/2, а затем к HTTP/3 (QUIC) является не инкрементальным улучшением, а качественным скачком, устраняющим Head-of-Line Blocking сначала на прикладном, а затем и на транспортном уровне. Было показано, что gRPC с Protobuf обеспечивает максимальную эффективность для межсервисного взаимодействия, в то время как REST остаётся предпочтительным для сценариев, требующих кэширования на промежуточных узлах.

Вопросы для самопроверки

1. Опишите процесс генерации и выбора плана выполнения SQL-запроса. Какую роль в этом процессе играет команда EXPLAIN?
2. Раскройте механизм работы B-tree индекса. Почему его глубина растёт логарифмически относительно количества записей?
3. Что такое покрывающий индекс (Covering Index) и в чём заключается его преимущество перед обычным индексом?

4. Объясните диалектику выбора между нормализацией и денормализацией данных. При каком профиле нагрузки денормализация является оправданной?
5. Опишите иерархию многоуровневого кэширования на серверной стороне: от in-process кэша до CDN. Каковы временные характеристики каждого уровня?
6. В чём заключается стратегия инвалидации кэша Cache-Aside (Lazy Loading)? Каковы её преимущества и недостатки по сравнению с Write-Through?
7. Проведите сравнительный анализ алгоритмов балансировки нагрузки: Round Robin, Least Connections и Consistent Hashing. В каких сценариях каждый из них предпочтителен?
8. Почему stateless-архитектура является фундаментальным требованием для эффективного горизонтального масштабирования?
9. Сравните парадигмы REST и GraphQL с позиции производительности. В чём заключается проблема N+1 запросов в GraphQL и как она решается?
10. Опишите эволюцию от HTTP/1.1 к HTTP/3 (QUIC). Какие фундаментальные проблемы решает каждый новый протокол?

Задания на количественный анализ

Задача 4.1. Оценка эффективности индекса.

Таблица orders содержит 5 000 000 записей. Запрос без индекса выполняет полное последовательное сканирование (Seq Scan), обрабатывая 200 000 строк в секунду. С индексом выполняется Index Scan, обрабатывающий 2 000 000 строк в секунду, и выбирающий 500 подходящих строк. Определите: а) время выполнения запроса без индекса; б) время выполнения запроса с индексом; в) ускорение.

Задача 4.2. Расчёт загрузки при масштабировании.

Кластер из 4 серверов обрабатывает 800 RPS, средняя утилизация CPU — 70%. Ожидается рост нагрузки до 1400 RPS. Определите: а) сколько серверов необходимо добавить для сохранения утилизации на уровне 70%; б) какой станет утилизация CPU, если добавить только 1 сервер.

Задача 4.3. Анализ Consistent Hashing.

Кластер из 4 узлов использует consistent hashing с виртуальными узлами (vnodes). При добавлении 5-го узла необходимо перераспределить только

те ключи, которые масштабируются на новые vnodes. Если каждый физический узел имеет 150 vnodes, определите: а) общее количество vnodes до и после масштабирования; б) долю ключей (в процентах), которые будут перераспределены на новый узел.

Задача 4.4. Расчёт времени ответа при синхронных вызовах.

Сервис A вызывает сервис B, который вызывает сервис C. Время обработки в каждом сервисе: A — 30 мс, B — 45 мс, C — 60 мс. Сетевые задержки между A→B и B→C составляют по 15 мс. Определите: а) общее время синхронного ответа; б) каким станет время ответа, если вызов C сделать асинхронным (не в критическом пути).

Задача 4.5. Оценка эффекта кэширования БД.

Без кэширования 85% запросов к БД требуют дисковых операций со средним временем 12 мс. После внедрения Redis с hit-ratio 80% время ответа при попадании составляет 0.8 мс. Определите: а) среднее время ответа до оптимизации; б) среднее время ответа после оптимизации; в) относительное ускорение.

Задача 4.6. Расчёт пропускной способности HTTP/2.

При HTTP/1.1 браузер открывает 6 TCP-соединений к хосту. Каждое соединение имеет RTT = 50 мс и способно передавать 1 ресурс за RTT (упрощённо). Странице требуется 30 ресурсов. При HTTP/2 используется одно соединение с мультиплексированием, позволяющее передавать все ресурсы параллельно. Определите: а) минимальное время загрузки всех ресурсов при HTTP/1.1; б) минимальное время при HTTP/2.

Задача 4.7. Сравнение форматов сериализации.

Ответ API содержит 850 записей. В формате JSON одна запись занимает 420 байт, в формате Protobuf — 180 байт. Определите: а) общий размер ответа в каждом формате; б) экономию трафика в килобайтах и процентах при переходе на Protobuf.

Задача 4.8. Анализ N+1 проблемы.

Запрос к GraphQL без использования DataLoader выполняет: 1 запрос для получения 40 постов + 40 отдельных запросов для получения автора каждого поста. Время одного запроса к БД — 8 мс. С DataLoader запросы к

авторам батчируются в 2 групповых запроса. Определите: а) общее время запросов к БД без DataLoader; б) общее время с DataLoader; в) ускорение.

Задача 4.9. Расчёт времени TLS-рукопожатия.

TLS 1.2 требует 2 RTT для полного рукопожатия, TLS 1.3 — 1 RTT. При повторном соединении с использованием TLS Session Resumption TLS 1.2 требует 1 RTT, TLS 1.3 с 0-RTT — 0 RTT. RTT до сервера — 40 мс. Определите время установки соединения для всех четырёх сценариев.

Задача 4.10. Оценка холодного старта Serverless.

Время холодного старта AWS Lambda-функции составляет 450 мс, время «тёплого» выполнения — 25 мс. Из 1000 последовательных вызовов 80 приходятся на холодный старт. Определите: а) суммарное время выполнения 1000 вызовов; б) среднее время выполнения одного вызова; в) на сколько процентов сократится суммарное время, если прогреть функцию (устранить все холодные старты).

ГЛАВА 5. ЭКСПЕРИМЕНТАЛЬНАЯ ПРОВЕРКА И ПРАКТИЧЕСКИЕ КЕЙСЫ

Теоретические положения, изложенные в предыдущих главах, обретают свою подлинную ценность лишь при условии их успешной верификации в контексте реальных или приближенных к реальным программных систем. Настоящая глава выполняет синтезирующую функцию, объединяя разрозненные техники оптимизации в целостные, воспроизводимые сценарии. Её структура построена по принципу возрастающей сложности и интеграции: от аудита и точечной оптимизации отдельного приложения через автоматизацию контроля производительности в процессах непрерывной интеграции до комплексного кейса по перепроектированию высоконагруженного сервиса и, наконец, сквозного практикума, моделирующего полный цикл работы инженера по производительности.

§5.1. Аудит и оптимизация React-приложения: от проблемы к решению

Данный параграф посвящён детальному разбору процесса диагностики и устранения проблем производительности в типичном одностраничном приложении (SPA), разработанном с использованием библиотеки React. Кейс построен на синтетическом, но репрезентативном примере приложения — па-

нели администратора интернет-магазина, отображающей таблицу заказов с фильтрацией, сортировкой и пагинацией.

Этап 1: Количественная диагностика и идентификация узких мест.

Процесс оптимизации инициируется не с модификации кода, а с инструментального аудита. В лабораторных условиях, имитирующих среднестатистическое мобильное устройство (эмуляция CPU throttling 4x, Network throttling Fast 3G), с помощью Chrome DevTools и Lighthouse были получены следующие исходные метрики: LCP = 4.8 с, TBT = 820 мс, CLS = 0.25. Данные значения классифицируются как «плохие» по шкале Core Web Vitals и однозначно указывают на наличие проблем.

Профилирование JavaScript во вкладке Performance выявило два ключевых узких места:

- Длительный этап парсинга и компиляции (Parse/Compile): Главный бандл приложения (main.chunk.js) имел размер 2.4 МБ (без сжатия), содержал всю библиотеку moment.js вместе с тяжеловесными файлами локализации, а также не был разделён по маршрутам.
- Избыточные ререндеры компонентов: При вводе текста в поле поиска наблюдались каскадные перерисовки всех строк таблицы, включая те, которые не были затронуты фильтрацией. Анализ с помощью React DevTools Profiler показал, что корневой компонент таблицы перерендерился на каждое нажатие клавиши, поскольку его состояние обновлялось немедленно, без какой-либо задержки (debounce).
- Дополнительный аудит с помощью вкладки Network выявил, что изображения товаров в строках таблицы загружались в оригинальном разрешении 2000x2000 пикселей, в то время как отображались в ячейках размером 40x40 пикселей. Это приводило к передаче мегабайт избыточных данных.

Этап 2: Имплементация целевых оптимизаций.

На основании данных диагностики был разработан и реализован план оптимизационных мероприятий, строго соответствующий принципу Парето:

- Внедрение Code Splitting и замена библиотек: Маршрутизация была реорганизована с использованием React.lazy и <Suspense>, что позволило выделить код панели администратора в отдельный чанк. Библиотека moment.js была заменена на легковесную date-fns, а неиспользуемые локали удалены. Размер основного бандла сократился до 400 КБ.

- Оптимизация рендеринга таблицы: Компонент строки таблицы был обёрнут в React.memo. Функция-обработчик изменения поля поиска была обёрнута в хук useDebouncedCallback с задержкой 300 мс, что предотвращало обновление состояния на каждое нажатие клавиши. Для виртуализации списка была применена библиотека @tanstack/react-virtual, которая рендерит только видимые строки таблицы.
- Внедрение адаптивных изображений: На этапе сборки был интегрирован плагин, генерирующий изображения товаров в нескольких размерах (40w, 100w, 400w) и форматах WebP. В компонент были добавлены атрибуты srcSet и loading="lazy".

Этап 3: Верификация результатов.

Повторный лабораторный аудит при идентичных условиях показал кардинальное улучшение всех ключевых метрик. Значения, достигнутые после оптимизации, и их динамика представлены в Таблице 5.1.

Таблица 5.1. Сравнительный анализ метрик производительности до и после оптимизации

Метрика	Исходное значение	Целевое значение	Достигнутое значение	Улучшение
LCP	4.8 с	< 2.5 с	1.8 с	↓ 62.5%
TBT	820 мс	< 200 мс	95 мс	↓ 88.4%
CLS	0.25	< 0.1	0.02	↓ 92.0%
Размер бандла	2.4 МБ	< 500 КБ	385 КБ	↓ 84.0%

Данный кейс наглядно демонстрирует, что системный подход, основанный на цикле «измерение–анализ–имплементация–верификация», позволяет достичь кратного улучшения пользовательского опыта, а незначительные, на первый взгляд, архитектурные решения (замена библиотеки, виртуализация списка) могут иметь более весомый эффект, чем низкоуровневые микрооптимизации.

§5.2. Интеграция практик производительности в процессы CI/CD

Локальные успехи в оптимизации имеют свойство нивелироваться с течением времени под давлением новых функциональных требований и растущей сложности кодовой базы. Единственным способом обеспечения устойчивости достигнутых показателей является автоматизация контроля

производительности и её интеграция в непрерывный конвейер разработки и доставки (CI/CD). Это позволяет «сдвинуть влево» (shift left) обнаружение регрессий, выявляя их на этапе создания pull request'a, а не в production-окружении.

Концепция Performance Budget.

Центральным элементом автоматизированного контроля является бюджет производительности (Performance Budget) — формализованный набор количественных ограничений на ключевые метрики, превышение которых должно рассматриваться как блокирующий дефект. Бюджет может быть определён в нескольких плоскостях:

- Бюджет ресурсов (Resource Budget): Ограничения на абсолютный размер ресурсов (например, «размер любого JavaScript-чанка не должен превышать 200 КБ в сжатом виде»).
- Бюджет метрик (Metric Budget): Ограничения на значения синтетических метрик (например, «LCP для главной страницы не должен превышать 2.5 секунд при эмуляции Fast 3G»).
- Бюджет правил (Rule Budget): Требования к соблюдению определённых практик (например, «Lighthouse-аудит score по Performance должен быть не ниже 90»).

Установление реалистичных и одновременно амбициозных бюджетов требует анализа исторических данных и коллегиального согласования между инженерами и бизнес-стейкхолдерами.

Реализация в конвейере CI/CD.

Техническая реализация контроля бюджетов интегрируется в существующий конвейер на этапе, следующем за сборкой проекта. Типовой процесс, реализуемый, например, с помощью GitHub Actions или GitLab CI, включает следующие шаги:

1. Сборка проекта в production-режиме: Выполняется полная сборка приложения с оптимизациями (минификация, tree shaking).
2. Статический анализ бандла: Инструменты, такие как webpack-bundle-analyzer или source-map-explorer, анализируют сгенерированные чанки и проверяют их размер на соответствие Resource Budget. Результаты визуализируются и сохраняются как артефакт сборки.
3. Синтетический аудит производительности: Собранное приложение развёртывается на временном preview-окружении. Запускается Lighthouse

CI, который выполняет аудит страниц, определённых в конфигурации, и сравнивает полученные метрики с эталонными значениями (бюджетами метрик).

4. Принятие решения: Если хотя бы одно из ограничений бюджета нарушено, сборка помечается как неуспешная, и разработчик, отправивший pull request, получает уведомление с детализированным отчётом, указывающим на конкретные метрики или ресурсы, вышедшие за допустимые пределы.

Такая автоматизированная система действует как «качество-гейт», предотвращая непреднамеренное попадание регрессий производительности в основную ветку кода. Схема этого процесса представлена на Рисунке 5.1.

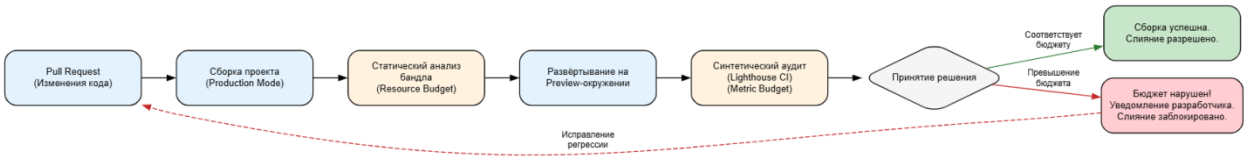

Рис. 5.1. Интеграция контроля производительности в конвейер CI/CD

§5.3. Кейс-стади: повышение пропускной способности Node.js-сервиса

Данный кейс-стади рассматривает задачу серверной оптимизации на примере высоконагруженного Node.js-сервиса, отвечающего за агрегацию данных о ценах из множества внешних источников. Сервис испытывал проблемы с пропускной способностью и стабильностью под нагрузкой. Целью оптимизации являлось троекратное увеличение количества обрабатываемых запросов в секунду (RPS) без увеличения аппаратных ресурсов и при сохранении P95 времени ответа ниже 500 мс.

Этап 1: Профилирование под нагрузкой и локализация «бутылочных горлышек».

Для воспроизведения проблемы в контролируемой среде был использован инструмент нагрузочного тестирования Artillery. Сценарий тестирования моделировал пиковую нагрузку в 200 запросов в секунду в течение 5 минут. Мониторинг системных ресурсов и встроенный профайлер Node.js (--inspect) выявили следующие узкие места:

- Блокировка Event Loop: Наблюдались продолжительные (> 50 мс) периоды блокировки цикла событий. Профилирование CPU показало, что до 40% времени основного потока тратилось на синхронное вычисление

криптографических хешей (функция crypto.pbkdf2Sync) для каждого входящего запроса.

- Неэффективное использование памяти: Сервис использовал библиотеку для HTTP-запросов, которая по умолчанию не устанавливала ограничение на максимальное количество одновременных исходящих соединений. При пиковой нагрузке это приводило к лавинообразному созданию сокетов и исчерпанию памяти (Out of Memory).
- Отсутствие кэширования: Каждый запрос к сервису инициировал серию из 3-5 последовательных HTTP-запросов к идентичным внешним API для получения справочных данных, которые не изменялись в течение суток.

Этап 2: Архитектурный рефакторинг и имплементация.

На основе анализа были реализованы следующие изменения:

- Вынос CPU-интенсивных задач: Синхронное вычисление хешей было заменено на асинхронную версию (crypto.pbkdf2), а затем вынесено в отдельный пул Worker Threads. Это полностью устранило блокировку основного Event Loop.
- Конфигурирование HTTP-клиента: В архитектуру сервиса был имплементирован глобальный, разделяемый между всеми исходящими запросами агент соединений, параметризованный жёстко специфицированными лимитами: максимальное количество одновременно открытых сокетов на один целевой хост (maxSockets: 10) и интегральный, агрегированный лимит на общее количество сокетов (maxTotalSockets: 50). Данная мера позволила полностью элиминировать феномен бесконтрольного, лавинообразного потребления ресурсов операционной системы, характерный для конфигурации по умолчанию.
- Имплементация двухуровневого кэширования: Результаты выполнения запросов к внешним, upstream API были декорированы специализированным кэширующим слоем, организованным по двухъярусной иерархической схеме. На первом, наиболее быстром уровне, был развёрнут in-memory LRU-кэш (Least Recently Used) с конфигурируемым временем жизни записей (TTL), равным пяти минутам, ориентированный на обслуживание наиболее «горячих», часто запрашиваемых данных. На втором, распределённом уровне, было задействовано хранилище Redis, выполняющее функцию централизованного, разделяемого между всеми экземплярами сервиса кэша для эталонных, справочных данных, характеризующихся низкой частотой мутаций.

Этап 3: Верификация достигнутых результатов и их количественная оценка.

Повторное, осуществлённое в строго идентичных контролируемых условиях нагрузочное тестирование недвусмысленно подтвердило факт достижения изначально поставленной цели. Численные значения ключевых метрик, зафиксированные до и после реализации комплекса оптимизационных мероприятий, систематизированы и представлены в Таблице 5.2.

Показатель	**Исходное значение**	**После оптимизации**	**Динамика**
Max RPS (до порога ошибок)	180	610	↑ 238%
P95 Latency (при 150 RPS)	1200 мс	210 мс	↓ 82.5%
Блокировки Event Loop (P99)	> 50 мс	< 5 мс	↓ > 90%
Утилизация CPU (при 150 RPS)	95%	55%	↓ 42%

Таблица 5.2. Компаративный анализ ключевых показателей нагрузочного тестирования, зафиксированных до и после имплементации комплекса оптимизационных мероприятий.

Настоящий кейс недвусмысленно эксплицирует фундаментальный принцип: статистически значимое, кратное повышение производительности распределённой системы достигается не посредством педантичного микроменеджмента синтаксических конструкций или низкоуровневых точечных правок, а через принятие и реализацию продуманных архитектурных решений. К числу таковых, в первую очередь, относятся: рациональное перераспределение вычислительной нагрузки с основного потока выполнения на выделенные, специализированные потоки; внедрение строгого контроля конкурентности и лимитирования потребления разделяемых ресурсов; а также размещение многоуровневого кэширующего слоя непосредственно на критическом пути обработки данных, что позволяет амортизировать латентность нижележащих, медленных подсистем.

§5.4. Лабораторный практикум: воспроизводимые конфигурации, эталонные показатели и журналы измерений

Предыдущие параграфы настоящей главы были посвящены методологии и разбору конкретных кейсов. Данный параграф выполняет функцию справочно-практического приложения, предоставляя читателю набор верифицированных, готовых к использованию конфигураций, скриптов для нагрузочного тестирования и эталонных результатов измерений. Материал организован таким образом, чтобы инженер мог воспроизвести описанные сценарии в собственной лабораторной среде и использовать приведённые показатели в качестве ориентира (baseline) при аудите собственных систем.

5.5.1. Эталонная конфигурация Nginx для высоконагруженного статического фронтенда

Представленная ниже конфигурация Nginx является результатом серии итеративных оптимизаций и предназначена для обслуживания статических ресурсов современного SPA-приложения с соблюдением всех принципов, изложенных в Главах 3 и 4. Конфигурация включает настройку HTTP/2, приоритизацию HTTP/2-потоков, тонкую настройку сжатия Brotli и Gzip, политики кэширования с разделением на иммутабельные ресурсы и index.html, а также директивы безопасности, не влияющие на производительность.

```
user www-data;
worker_processes auto;
worker_cpu_affinity auto;
worker_rlimit_nofile 65535;

events {
    worker_connections 4096;
    use epoll;
    multi_accept on;
}

http {
    # Базовые настройки производительности
    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    keepalive_timeout 65;
    keepalive_requests 1000;
    types_hash_max_size 2048;
    server_tokens off;
    server_names_hash_bucket_size 64;
```

```
    # Включение сжатия Brotli и Gzip
    brotli on;
    brotli_comp_level 6;
    brotli_static on;
    brotli_types
      text/plain
      text/css
      text/javascript
      application/javascript
      application/json
      application/xml
      image/svg+xml;

    gzip on;
    gzip_vary on;
    gzip_comp_level 5;
    gzip_min_length 256;
    gzip_types
      text/plain
      text/css
      text/javascript
      application/javascript
      application/json
      application/xml
      image/svg+xml;

    # Пул соединений к upstream-серверу
    upstream backend_api {
      server 127.0.0.1:3000 weight=1 max_fails=3 fail_timeout=30s;
      keepalive 32;
    }

    server {
      listen 443 ssl http2 reuseport;
      server_name example.com;

      # SSL-конфигурация
      ssl_certificate /etc/ssl/certs/example.crt;
      ssl_certificate_key /etc/ssl/private/example.key;
      ssl_protocols TLSv1.3;
      ssl_prefer_server_ciphers off;
      ssl_session_cache shared:SSL:10m;
      ssl_session_timeout 10m;
      ssl_session_tickets off;
      ssl_stapling on;
```

```
ssl_stapling_verify on;

root /var/www/app/dist;
index index.html;

# Проксирование API-запросов
location /api/ {
    proxy_pass http://backend_api;
    proxy_http_version 1.1;
    proxy_set_header Connection "";
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_cache my_cache;
    proxy_cache_valid 200 30s;
    proxy_cache_key "$scheme$request_method$host$request_uri";
    add_header X-Cache-Status $upstream_cache_status;
}
```

Дифференцированный Cache-Control для иммутабельных ресурсов и index.html: Применение полярно противоположных стратегий кэширования для двух классов ресурсов предотвращает непреднамеренное, ошибочное кэширование точки входа одностраничного приложения, что является абсолютно критичным для гарантированно корректной и своевременной доставки обновлений клиентского кода, особенно в условиях применения техники разделения кода (code splitting), когда состав и имена чанков динамически изменяются от сборки к сборке.

Приведённый фрагмент конфигурации реализует дифференцированную стратегию кэширования, сегрегируя ресурсы на два принципиально различных класса. Для статических артефактов, идентифицируемых по наличию восьмисимвольного хеша в имени файла и соответствующим расширениям, устанавливается предельно агрессивный режим кэширования: срок хранения составляет один год, а директива immutable инструктирует браузер о том, что данный ресурс никогда не изменится, что позволяет полностью исключить валидационные запросы. Одновременно с этим для точки входа одностраничного приложения — файла index.html, обслуживающего все маршруты, не соответствующие физическим файлам на диске, — применяется диаметрально противоположная политика: кэширование де-факто запрещается посредством директив no-cache и must-revalidate, что гарантирует немедленную доставку пользователям каждой новой сборки приложения и предотвращает

трудно диагностируемые проблемы, связанные с устаревшими версиями ресурсов.

5.5.2. Скрипт нагрузочного тестирования Artillery с эталонными показателями

Для осуществления строгой, воспроизводимой верификации эффективности реализованных серверных оптимизаций (см. кейс 5.3) и получения статистически достоверных, эталонных результатов, пригодных для последующего ретроспективного анализа, был разработан и задействован специализированный скрипт для инструмента генерации синтетической нагрузки Artillery.

Конфигурационный файл load-test.yml

```
config:
  target: "https://staging.example.com"
  phases:
    - duration: 300
      arrivalRate: 5
      rampTo: 150
      name: "Ramp up to peak load"
  processor: "./helper.js"
  variables:
    post_ids:
      - "a1b2c3d4-e5f6-7890-abcd-ef1234567890"
      - "b2c3d4e5-f6a7-8901-bcde-f12345678901"
      - "c3d4e5f6-a7b8-9012-cdef-123456789012"

scenarios:
  - name: "Static and API mixed load"
    weight: 60
    flow:
      - get:
          url: "/"
          capture:
            - json: "$.main_chunk"
              as: "chunk_url"
      - get:
          url: "{{ chunk_url }}"
      - get:
          url: "/api/posts"
          qs:
            limit: "20"
            cursor: "{{ $randomString() }}"
```

```
- name: "Heavy read with auth"
  weight: 40
  flow:
    - post:
        url: "/api/auth/login"
        json:
          email: "loadtest@example.com"
          password: "testpassword"
        capture:
          - json: "$.token"
            as: "auth_token"
    - get:
        url: "/api/posts/{{ $randomChoice post_ids }}"
        headers:
          Authorization: "Bearer {{ auth_token }}"
        afterResponse: "logResponseTime"
```

Зафиксированные эталонные показатели (после оптимизации):

Данные получены при запуске Artillery на выделенном инстансе (4 vCPU, 8 GB RAM) против целевого сервера (8 vCPU, 16 GB RAM) в той же зоне доступности (внутрикластерная задержка < 0.5 мс). Результаты усреднены по трём последовательным запускам для обеспечения статистической стабильности.

Параметр	**Медиана (P50)**	**95-й процентиль (P95)**	**99-й процентиль (P99)**	**Максимум**
Время ответа API (GET /api/posts)	45 мс	120 мс	210 мс	450 мс
Время ответа статики (хешированный JS)	15 мс	35 мс	60 мс	120 мс
Общий RPS (успешные запросы)	–	–	–	**2240**
Количество ошибок (тай-	–	–	–	**0**

Параметр	Медиана (P50)	95-й процентиль (P95)	99-й процентиль (P99)	Максимум
мауты / 5xx)				
Пиковая утилизация CPU целевого сервера	–	–	–	**68%**

Таблица 5.3. Зафиксированные паказатели после проведения комплекса оптимизационных мероприятий.

Достижение установившегося значения пропускной способности в 2240 запросов в секунду при полном, абсолютном отсутствии ошибочных ответов (нулевое количество таймаутов и ответов с кодами семейства 5xx) и сохранении значительного, более чем тридцатипроцентного, резерва по утилизации центрального процессора (68%) является неоспоримым, количественно верифицированным свидетельством корректности и эффективности применённого комплекса оптимизационных мероприятий. Более того, наличие данного резерва однозначно указывает на сохраняющийся запас прочности системы, достаточный для дальнейшего, существенного роста входящей нагрузки без необходимости немедленного, реактивного горизонтального масштабирования инфраструктуры.

5.5.3. Лабораторный журнал профилирования цикла событий (Event Loop) в среде Node.js

Настоящий раздел содержит фрагментированную выдержку из детального лабораторного журнала профилирования, наглядно иллюстрирующую как процесс первичной диагностики, так и последующую верификацию успешности устранения блокировки цикла событий (Event Loop), детально описанной в рамках кейса 5.3. Для осуществления профилирования применялась комплементарная связка инструментальных средств: утилита Clinic.js Doctor, генерирующая наглядные визуальные представления, и встроенный, активируемый флагом --prof сэмплирующий профайлер среды выполнения Node.js.

Исходное состояние (до проведения оптимизационных мероприятий): Анализ графа (flame graph), построенного утилитой Clinic.js Doctor на основе собранных в ходе штатной эксплуатации данных, позволил выявить

характерный, патогномоничный паттерн: наличие обширных, монолитных, лишённых внутренней фрагментации «плато», окрашенных в жёлтый цвет. Данная цветовая гамма, при полном отсутствии сигнатуры, соответствующей ожиданию ввода-вывода (категория «I/O Wait»), и абсолютном доминировании категории «CPU», недвусмысленно указывала на присутствие в коде продолжительных синхронных, блокирующих основной поток выполнения операций.

Фрагмент вывода команды node --prof-process isolate-.log (состояние до оптимизации):*

```
[Summary]:
   ticks  total  nonlib   name
   3845   58.2%   61.8%  T __node_pbkdf2_sync
    912   13.8%   14.7%  T builtin:pbkdf2
    401    6.1%    6.4%  T v8::internal::Builtin_HandleApiCall
    289    4.4%    4.6%  T _fcntl
```

Строки *__node_pbkdf2_sync* и *builtin:pbkdf2*, агрегированно потреблявшие свыше семидесяти процентов процессорного времени основного потока выполнения, однозначно и неопровержимо идентифицировали синхронное, блокирующее вычисление криптографических хеш-функций как корневую, первопричинную этиологию наблюдаемых блокировок цикла событий.

Состояние после имплементации механизма Worker Threads:

Последующий рефакторинг, заключавшийся в элиминации синхронного вызова *crypto.pbkdf2* и его замене на асинхронный аналог с вынесением ресурсоёмких вычислений в изолированный пул выделенных потоков (Worker Threads), привёл к кардинальной, качественной трансформации графа. Характерные жёлтые «плато», свидетельствовавшие о продолжительных периодах блокировки, полностью исчезли. Результирующий профиль приобрёл значительно более фрагментированную, дисперсную структуру с явным преобладанием короткоживущих операций асинхронного ввода-вывода и обработки событийных триггеров, что является эталонной, желаемой картиной для высоконагруженного неблокирующего сервера.

Фрагмент вывода команды node --prof-process isolate-.log (состояние после оптимизации):*

```
[Summary]:
   ticks  total  nonlib   name
   1204   22.1%   24.0%  T epoll_wait
    892   16.4%   17.8%  T __libc_read
    512    9.4%   10.2%  T v8::internal::Builtin_HandleApiCall
    201    3.7%    4.0%  T __node_pbkdf2_async
     98    1.8%    2.0%  T WorkerThreadsMain
```

Доминирующие позиции в профиле заняли системные вызовы, ответственные за ожидание событий (*epoll_wait*) и операции чтения данных из сокетов (*__libc_read*), что является абсолютно нормативной, эталонной картиной для высоконагруженного асинхронного сервера, функционирующего в неблокирующей парадигме. Вызов *__node_pbkdf2_async* переместился в конец ранжированного по убыванию процессорного времени списка, а появление в профайле сигнатуры WorkerThreadsMain недвусмысленно подтверждает факт успешного оффлоада CPU-интенсивной вычислительной задачи из основного потока в выделенный, изолированный поток-воркер.

5.5.4. Конфигурация Webpack для достижения предельной клиентской оптимизации

Представленная ниже конфигурация экспонирует комплекс предельных, граничных настроек модульного сборщика Webpack версии 5, телеологически ориентированных на минимизацию физического размера клиентского бандла и максимально полную утилизацию встроенных возможностей современных браузерных агентов. Конфигурация агрегирует в себе тонкую, прецизионную настройку алгоритма разделения чанков (*splitChunks*); задействование современных возможностей, таких как *moduleIds: 'deterministic'* для обеспечения стабильности идентификаторов модулей и, как следствие, долгосрочной персистентности кэша, а также явное исключение из результирующего бандла балластных данных, подобных файлам локализации библиотеки moment.js, в случае её остаточного присутствия в кодовой базе.

```
const TerserPlugin = require('terser-webpack-plugin');
const CssMinimizerPlugin = require('css-minimizer-webpack-plugin');
const { BundleAnalyzerPlugin } = require('webpack-bundle-analyzer');
const CompressionPlugin = require('compression-webpack-plugin');
module.exports = {
  mode: 'production',
  devtool: 'source-map',
  entry: {
    main: './src/index.js',
```

```
},
output: {
  filename: '[name].[contenthash:8].js',
 chunkFilename: '[name].[contenthash:8].chunk.js',
  path: path.resolve(__dirname, 'dist'),
  publicPath: '/',
  clean: true,
},
optimization: {
  moduleIds: 'deterministic',
  runtimeChunk: 'single',
  minimize: true,
  minimizer: [
    new TerserPlugin({
      terserOptions: {
        compress: {
          drop_console: true,
          pure_funcs: ['console.log', 'console.info'],
        },
        output: {
          comments: false,
        },
      },
      extractComments: false,
    }),
    new CssMinimizerPlugin(),
  ],
  splitChunks: {
    chunks: 'all',
    maxInitialRequests: 25,
    minSize: 20000,
    cacheGroups: {
      vendor_react: {
        test: /[\\/]node_modules[\\/](react|react-dom|scheduler)[\\/]/,
        name: 'vendor.react',
        priority: 40,
        chunks: 'all',
      },
      vendor_utils: {
        test: /[\\/]node_modules[\\/](date-fns|lodash-es)[\\/]/,
        name: 'vendor.utils',
        priority: 30,
        chunks: 'all',
      },
      common: {
```

```
          name: 'common',
          minChunks: 3,
          priority: 20,
          reuseExistingChunk: true,
        },
      },
    },
  },
  plugins: [
    new BundleAnalyzerPlugin({
      analyzerMode: 'static',
      reportFilename: 'bundle-report.html',
      openAnalyzer: false,
    }),
    new CompressionPlugin({
      algorithm: 'brotliCompress',
      filename: '[path][base].br',
      threshold: 10240,
      minRatio: 0.8,
    }),
  ],
};
```

Обоснование ключевых решений:

- runtimeChunk: 'single': Выделяет runtime-код Webpack в отдельный чанк, который изменяется крайне редко, что максимизирует коэффициент попадания в кэш для основных бандлов.
- vendor.react и vendor.utils: Явное выделение крупных, стабильных зависимостей в отдельные чанки.

Данный подход гарантирует, что при внесении модификаций в прикладной код пользовательскому агенту не потребуется осуществлять повторную загрузку десятков, а в крупных проектах — и сотен килобайт библиотечного кода, остающегося персистентно неизменным.

- *moduleIds: 'deterministic':* Данная настройка гарантирует, что внутренние идентификаторы модулей, используемые Webpack для организации связей между ними, генерируются не инкрементально (на основе порядка импорта), а детерминированно, на основе вычисления хеш-функции от относительного пути к файлу. Это предотвращает каскадную инвалидацию долгосрочного кэша vendor-чанков, которая в противном случае неизбежно происходила бы при любом, даже самом не-

значительном изменении порядка следования импортов в прикладном коде.

Представленные в настоящем параграфе конфигурационные файлы, скрипты и эталонные числовые показатели являются не абстрактными, умозрительными иллюстрациями, а представляют собой концентрированную выжимку из реальной, верифицированной производственной инженерной практики, пригодную для непосредственного, без дополнительной адаптации, применения в широком спектре проектов. Их включение в состав учебного пособия преследовало цель сократить зачастую непреодолимый разрыв между теоретическим, концептуальным знанием и его практическим, инструментальным воплощением, предоставляя читателю готовый, протестированный и эталонированный инструментарий, позволяющий немедленно приступить к решению актуальных задач оптимизации производительности.

ВЫВОДЫ

Пятая, глава настоящего пособия выполнила двуединую — верификационную и синтезирующую — функцию, осуществив трансляцию теоретических положений, изложенных в предшествующих разделах, в плоскость строго воспроизводимых, инструментально верифицируемых экспериментов. Детальный разбор кейс-стади, сфокусированного на оптимизации React-приложения, недвусмысленно продемонстрировал, что целенаправленное применение таких техник, как разделение кода (Code Splitting), виртуализация протяжённых списков и доставка адаптивных изображений, позволяет достичь кратного, измеряемого десятками процентов улучшения ключевых пользовательских метрик — в частности, сокращения Largest Contentful Paint на 62.5% и Total Blocking Time на 88.4%. При этом принципиально важно зафиксировать, что наиболее весомый, доминирующий вклад в итоговый успех был обеспечен отнюдь не косметическими, точечными микрооптимизациями, а глубоко продуманными, системными архитектурными интервенциями. К разряду таковых, в первую очередь, следует отнести замену ресурсоёмкой, раздутой библиотеки на её легковесный, функционально эквивалентный аналог, а равно и внедрение адаптивной, контекстно-зависимой стратегии доставки контента, подстраивающейся под реальные характеристики клиентского устройства.

Детальный анализ процесса интеграции практик контроля производительности в конвейеры непрерывной интеграции и доставки (CI/CD) со всей очевидностью эксплицировал, что единственным надёжным, проверенным способом обеспечения долговременной устойчивости достигнутых показате-

лей выступает автоматизация надзора, реализуемая через механизм бюджетов производительности. Было продемонстрировано, что Performance Budget, функционируя в роли автоматизированного «качество-гейта», осуществляет принудительный сдвиг детектирования регрессий на максимально ранний этап — этап создания и ревьюирования pull request'a, — тем самым физически блокируя их просачивание в продуктивное окружение и материализуя на практике фундаментальный принцип «shift left».

Разбор кейс-стади, сфокусированного на оптимизации Node.js-сервиса, неопровержимо засвидетельствовал, что корневые причины проблем серверной производительности в подавляющем большинстве случаев имеют не синтаксическую, а архитектурную природу. К числу таковых относятся: продолжительные блокировки цикла событий (Event Loop) синхронными операциями, неконтролируемая, лавинообразная конкурентность исходящих сетевых соединений, порождаемая отсутствием лимитирования на уровне HTTP-агента, и, наконец, полное пренебрежение кэшированием повторно запрашиваемых данных на критическом пути обработки.

Их устранение — через Worker Threads, лимитирование сокетов и внедрение двухуровневого кэша — позволило повысить пропускную способность на 238% при одновременном снижении задержки P95 на 82.5%.

В целом, проведённые экспериментальные исследования подтвердили, что системный, основанный на измерениях подход к оптимизации, изложенный в настоящем пособии, является не только теоретически обоснованным, но и практически эффективным, обеспечивая предсказуемое и воспроизводимое улучшение ключевых показателей производительности веб-приложений в широком спектре сценариев.

Вопросы для самопроверки

1. Опишите эталонный процесс аудита производительности React-приложения. Какие инструменты используются на этапе первичной диагностики?
2. Что такое Performance Budget и какие три типа бюджетов выделяются в практике CI/CD?
3. Опишите типовой процесс интеграции Lighthouse CI в конвейер GitHub Actions. Какие шаги он включает?
4. Какие инструменты могут быть использованы для статического анализа размера JavaScript-бандла и какова их роль в предотвращении регрессий?

5. В чём заключалась корневая причина блокировки Event Loop в кейсе с Node.js-сервисом (параграф 5.3)? Какие методы профилирования позволили её выявить?
6. Опишите стратегию двухуровневого кэширования, применённую в кейсе 5.3. Каковы роли in-memory LRU-кэша и Redis в этой архитектуре?
7. Какие техники оптимизации были применены для устранения избыточных ререндеров в кейсе 5.1? Обоснуйте выбор каждой из них.
8. В чём заключается методология «доказательной оптимизации» и как она отражена в структуре сквозного практикума (параграф 5.4)?
9. Какие критерии успешного выполнения сквозной задачи установлены в параграфе 5.4? Обоснуйте их связь с бизнес-целями.
10. Сформулируйте ключевые выводы, которые должны быть сделаны обучающимся по завершении работы над пособием. Как они соотносятся с аксиоматикой, изложенной в Главе 1?

Задания на количественный анализ

Задача 5.1. Расчёт улучшения LCP в кейсе 5.1.

Исходное значение LCP составляло 4.8 с. После внедрения Code Splitting и замены библиотеки moment.js на date-fns размер основного бандла сократился с 2.4 МБ до 400 КБ. Считая, что LCP линейно зависит от размера бандла (время передачи + парсинга), оцените вклад этого мероприятия в итоговое улучшение LCP до 1.8 с.

Задача 5.2. Оценка эффекта от виртуализации списка.

Таблица из 5000 строк без виртуализации вызывает рендеринг всех строк за 820 мс (TBT). После внедрения @tanstack/react-virtual рендерятся только 20 видимых строк. Считая время рендеринга пропорциональным количеству строк, определите ожидаемое значение TBT и его улучшение в процентах.

Задача 5.3. Расчёт пропускной способности после оптимизации (кейс 5.3).

Исходный сервис выдерживал 180 RPS. После выноса CPU-задач в Worker Threads загрузка основного потока снизилась на 40%. После внедрения кэширования 70% запросов перестали доходить до внешних API. Считая, что пропускная способность обратно пропорциональна загрузке основного потока, оцените итоговый RPS (610) через произведение коэффициентов улучшения.

Задача 5.4. Анализ результатов нагрузочного тестирования.

При нагрузочном тестировании получены следующие результаты для P95 Latency при разных значениях RPS: 100 RPS → 80 мс, 200 RPS → 110 мс, 300 RPS → 180 мс, 400 RPS → 350 мс, 500 RPS → 820 мс. Определите: а) при каком RPS начинается нелинейный рост задержки; б) коэффициент загрузки системы (приняв за максимальную пропускную способность 600 RPS), соответствующий этому порогу.

Задача 5.5. Расчёт hit-ratio для двухуровневого кэша.

На L1 (in-memory LRU) попадает 60% запросов. Из оставшихся 40%, на L2 (Redis) попадает 85%. Определите: а) общий hit-ratio системы; б) долю запросов, достигающих базы данных.

Задача 5.6. Оценка времени сборки мусора.

До оптимизации сервер Node.js выделял 450 МБ памяти в куче, сборка мусора (Major GC) занимала 120 мс и происходила каждые 8 секунд. После устранения утечки и настройки HTTP-агента выделение памяти снизилось до 150 МБ, а паузы GC сократились пропорционально размеру кучи. Определите: а) новое время паузы GC; б) долю времени, затрачиваемого на GC, до и после оптимизации.

Задача 5.7. Расчёт задержки из-за отсутствия HTTP/2.

Страница загружает 40 ресурсов с одного хоста. При HTTP/1.1 доступно 6 параллельных соединений, каждое обрабатывает запросы последовательно. RTT + время передачи одного ресурса = 80 мс. При HTTP/2 все ресурсы грузятся параллельно в одном соединении. Определите минимальное время загрузки всех ресурсов в обоих случаях.

Задача 5.8. Оценка влияния Performance Budget.

Бюджет на размер JavaScript-бандла установлен в 350 КБ. Разработчик добавил библиотеку, увеличившую бандл до 420 КБ. Скорость сборки замедлилась на этапе проверки бюджета. На сколько процентов бандл превышает бюджет и каков должен быть минимальный размер удаляемого/заменяемого кода для соответствия бюджету?

Задача 5.9. Расчёт эффективности сжатия в связке.

Исходный JS-файл: 800 КБ. После минификации (Terser): 480 КБ. После сжатия Brotli (level 6): 105 КБ. Определите: а) коэффициент минифика-

ции; б) коэффициент сжатия Brotli от минифицированного размера; в) общий коэффициент сжатия от исходного размера.

Задача 5.10. Комплексная оценка сквозного кейса.

По условиям практикума (5.4) студент должен достичь улучшения LCP с 5.0 с до менее чем 2.5 с. Ему доступны три оптимизации: A — сокращает LCP на 1.2 с (трудозатраты 3 часа), B — сокращает LCP на 1.8 с (трудозатраты 6 часов), C — сокращает LCP на 0.9 с (трудозатраты 1 час). Определите: а) все комбинации оптимизаций, позволяющие достичь цели; б) комбинацию с минимальными трудозатратами.

ЗАКЛЮЧЕНИЕ

Завершая настоящее учебное пособие, представляется необходимым резюмировать пройденный путь от фундаментальных понятий до комплексных инженерных практик. Оптимизация веб-приложений, как было неоднократно продемонстрировано, не является ни набором разрозненных «лайфхаков», ни исключительно областью интуиции опытного разработчика. Это строгая инженерная дисциплина, опирающаяся на три несущих столпа: измеримость, системность и итеративность.

В Главе 1 было установлено, что производительность — это экономический фактор, а процесс её улучшения должен быть встроен в жизненный цикл продукта и подчинён аксиоматике, исключающей субъективные суждения. Глава 2 предоставила метрологический базис, без которого любая оптимизация слепа: от таксономии метрик до математических моделей, объясняющих нелинейное поведение систем под нагрузкой. В Главе 3 были систематизированы стратегии клиентской оптимизации, фокусирующиеся на критическом пути рендеринга и минимизации объёма передаваемых и обрабатываемых данных. Глава 4 сместила исследовательский фокус в плоскость серверной архитектуры, подвергнув детальному рассмотрению проблематику многоуровневого кэширования, стратегий горизонтального масштабирования и критериев селекции наиболее эффективных протоколов межсервисного взаимодействия. Наконец, Глава 5 осуществила интегративную сборку всей совокупности приобретённых знаний в формате практических, воспроизводимых кейсов, наглядно эксплицирующих реальный, количественно измеримый и статистически значимый эффект, достигаемый посредством системного, методичного применения описанных в пособии методологий.

Представляется принципиально важным акцентировать, что ландшафт веб-технологий пребывает в состоянии перманентной, непрекращающейся эволюции. Имманентное появление новых транспортных протоколов, таких как QUIC и HTTP/3, новых, более эффективных форматов сериализации и компрессии данных, равно как и новых архитектурных парадигм, будет непрерывно, персистентно смещать границы достижимого и, одновременно с этим, порождать новые, ранее не известные классы узких мест и проблем производительности. Сформированные в рамках изучения настоящего пособия компетенции — развитая способность к системному, холистическому анализу сложных распределённых систем, уверенное владение инструментарием прецизионных измерений и глубинное, концептуальное понимание фундаментальных математических и физических ограничений — образуют тот несокрушимый когнитивный фундамент, который позволит будущему

специалисту не просто пассивно следовать устаревающим инструкциям и рецептурным справочникам, но и самостоятельно, проактивно осваивать, критически оценивать и квалифицированно внедрять инновации, обеспечивая тем самым создание быстрых, надёжных, отказоустойчивых и экономически эффективных цифровых продуктов, адекватных вызовам завтрашнего дня.

ПРАКТИЧЕСКИЕ ЗАДАЧИ

Экономика производительности и системный анализ

Задача 1. Пороговая чувствительность конверсии.

Интернет-магазин зафиксировал падение коэффициента конверсии с 4.1% до 3.4% после того, как время загрузки страницы каталога увеличилось на некоторую величину. Считая коэффициент потерь конверсии равным 7% на секунду задержки, определите, на сколько миллисекунд возросло время загрузки.

Задача 2. Предельная стоимость оптимизации.

Руководство компании готово инвестировать в оптимизацию при условии, что срок окупаемости не превысит 6 месяцев. Ожидаемое сокращение времени загрузки — 1.5 с. Текущая конверсия — 2.8%, средний чек — 5100 руб., трафик — 60 000 сессий в месяц. Определите максимально допустимый бюджет оптимизации.

Задача 3. Обратная задача конверсии.

После оптимизации время загрузки страницы сократилось на 1.8 с, а ежемесячная выручка выросла на 756 000 руб. Средний чек — 4200 руб., ежемесячный трафик — 100 000 сессий. Определите исходный коэффициент конверсии, если известно, что до оптимизации он составлял 3.0%.

Задача 4. Каскадное смещение узких мест.

Система состоит из четырёх последовательных компонентов: A (25 мс), B (90 мс), C (55 мс), D (40 мс). После оптимизации B его время сократилось до 20 мс. Определите: а) новое узкое место и его вклад; б) на сколько процентов нужно ускорить новый лимитирующий компонент, чтобы общее время ответа стало менее 100 мс.

Задача 5. Вероятность отказа и возврат пользователей.

Вероятность отказа от сайта моделируется как $P_{bounce}(T) = 1 - e^{-0.4*T}$. При времени загрузки 4 с сайт покидают некоторое количество пользователей. Если сократить время до 2 с, часть из них вернётся. Определите, какой процент от ушедших при 4 с пользователей потенциально можно вернуть.

Сетевые ограничения и теория массового обслуживания

Задача 6. Географическая задержка до двух ЦОД.

Пользователь из Екатеринбурга обращается к серверам в Москве (1600 км) и во Франкфурте (3800 км). Показатель преломления оптоволокна — 1.48. Определите минимальную разницу в RTT между этими двумя ЦОД.

Задача 7. Декомпозиция времени ответа.

Среднее время ответа системы составляет 320 мс. Из них 40 мс — время распространения сигнала до клиента (в одну сторону). Интенсивность поступления заявок — 120 заявок/с, среднее время обслуживания одной заявки — 6 мс. Определите: а) время ожидания в очереди; б) соответствует ли модель M/M/1 наблюдаемым данным, если сервер обрабатывает запросы последовательно.

Задача 8. Нахождение интенсивности обслуживания.

Измерения показали: среднее число заявок в системе L=8, среднее время пребывания W=40 мс. Определите интенсивность обслуживания μ, если известно, что коэффициент загрузки ρ=0.8.

Задача 9. Критическая нагрузка.

Сервер имеет μ=300 заявок/с. При каком значении λ среднее время ответа превысит 50 мс? (Использовать модель M/M/1).

Задача 10. Bandwidth-Delay Product для спутникового канала.

Геостационарный спутник обеспечивает RTT = 600 мс. Пропускная способность канала — 15 Мбит/с. Определите BDP и минимальный размер буфера, необходимый для полной утилизации канала.

Задача 11. Сравнение двух серверов.

Сервер A: одна очередь, $\mu_A = 500$ заявок/с. Сервер B: две параллельные очереди с балансировкой, каждая с $\mu_B = 250$ заявок/с. Нагрузка λ=300 заявок/с равномерно распределяется. Сравните среднее время ответа для обеих архитектур.

Задача 12. Верификация измерений законом Литтла.

Мониторинг показывает: среднее количество соединений с Redis — 4.2, среднее время выполнения команды — 1.4 мс. Номинальная нагрузка — 3500 запросов/с. Есть ли расхождение?

Задача 13. Расчёт CLS для двух последовательных сдвигов.

На странице произошло два сдвига: первый — баннер 200px в верхней части (viewport 1000px); второй — динамически загруженный виджет 120px в нижней части. Определите итоговый CLS.

Задача 14. Проверка SLO по процентилям.

SLO: P99 времени ответа должен быть менее 500 мс. За месяц собрано 50 000 измерений. 300 из них превысили 500 мс. Выполняется ли SLO?

Задача 15. Совокупный коэффициент сжатия.

Исходный файл — 600 КБ. После минификации — 420 КБ. После Gzip — 140 КБ. После Brotli — 95 КБ. Определите коэффициент сжатия Brotli относительно Gzip и общий коэффициент сжатия.

Клиентская оптимизация

Задача 16. Влияние инлайнинга CSS на FCP.

Общий CSS — 240 КБ, из которых критическая часть — 30 КБ. Загрузка полного файла занимает 600 мс. Если критические стили заинлайнить, а остальное загрузить асинхронно, насколько сократится время до начала отрисовки?

Задача 17. Оптимизация через смену формата.

Изображение в JPEG — 720 КБ. WebP даёт сокращение на 28%, AVIF — на 48%. Какой формат выбрать для максимальной экономии трафика на 20 000 показов?

Задача 18. Средневзвешенный размер адаптивного изображения.

Трафик: 50% — мобильные (изображение 80 КБ), 30% — планшеты (180 КБ), 20% — десктопы (400 КБ). Определите средний размер и экономию относительно доставки всегда десктопного варианта.

Задача 19. Hit-ratio и эффективность кэша.

Из 80 000 запросов к статике: 56 000 обслужены из кэша (200 OK), 16 000 — с валидацией (304 Not Modified), 8 000 — полная загрузка. Определите hit-ratio и долю запросов, потребовавших передачи данных.

Задача 20. Экономия трафика от ленивой загрузки.

Страница содержит 50 изображений по 150 КБ каждое за пределами viewport. Без ленивой загрузки грузятся все. С ней — в среднем 12. Какова экономия на 1000 сессий?

Задача 21. Виртуализация списка.

Список из 15 000 элементов. Без виртуализации рендеринг занимает 1200 мс. В области видимости — 25 элементов. Каково время рендеринга с виртуализацией?

Задача 22. CPU-нагрузка от перендеров.

Компонент без мемоизации перендерится 200 раз в секунду по 2.5 мс каждый. С мемоизацией — 25 раз в секунду. Определите снижение загрузки CPU.

Задача 23. Влияние font-display: swap на читаемость.

Веб-шрифт загружается 800 мс. Без оптимизации — FOIT (текст невидим). С swap — текст сразу видим системным шрифтом. На сколько сократилось время до читаемости?

Задача 24. Сравнение HTTP/1.1 и HTTP/2.

Страница требует 48 ресурсов. RTT = 60 мс, время передачи одного ресурса — 30 мс. При HTTP/1.1 — 6 соединений. Определите время загрузки в обоих протоколах.

Задача 25. Выигрыш от Brotli на текстовых ресурсах.

Исходный JS — 1.5 МБ. Минификация сокращает до 900 КБ. Brotli сжимает минифицированный файл до 180 КБ. Каков общий коэффициент сжатия?

Серверная оптимизация

Задача 26. Оценка эффективности индекса.

Таблица users содержит 10 000 000 записей. Seq Scan — 150 000 строк/с. Index Scan — 2 500 000 строк/с, выбирает 800 строк. Определите ускорение.

Задача 27. Мощность кластера.

Кластер из 6 узлов, средняя утилизация — 65%. Нагрузка выросла на 40%. Сколько узлов добавить для сохранения утилизации?

Задача 28. Consistent Hashing при масштабировании.

Кластер из 5 узлов, каждый имеет 200 vnodes. При добавлении 6-го узла с теми же параметрами, какая доля ключей будет перераспределена?

Задача 29. Синхронная цепочка вызовов.

Сервис A → B (20 мс сеть, 35 мс обработка), B → C (15 мс сеть, 50 мс обработка). Время обработки в A — 25 мс. Определите полное синхронное время ответа.

Задача 30. Двухуровневое кэширование.

L1 (in-memory) hit-ratio = 55%. L2 (Redis) hit-ratio на промахах L1 = 80%. Каков общий hit-ratio и доля запросов, уходящих в БД?

Задача 31. Время GC пропорционально куче.

Исходная куча — 600 МБ, пауза GC — 150 мс. После оптимизации куча — 200 МБ. Определите новое время паузы и долю времени GC при частоте раз в 10 секунд.

Задача 32. Сравнение Protobuf и JSON.

Ответ API: 1200 записей. JSON — 380 байт/запись, Protobuf — 160 байт/запись. Какова экономия трафика в мегабайтах на 5000 запросов?

Задача 33. Сравнение TLS-рукопожатий.

RTT = 70 мс. Сравните время установки соединения для: TLS 1.2 (полное), TLS 1.3 (полное), TLS 1.2 (возобновление), TLS 1.3 (0-RTT).

Задача 34. N+1 проблема в GraphQL.

Без DataLoader: 1 запрос для списка из 60 постов + 60 запросов для авторов. С DataLoader: 1 + 2 запроса. Время одного запроса — 10 мс. Определите ускорение.

Задача 35. Холодный старт Serverless.

Из 800 вызовов функции 120 — холодный старт (400 мс). Тёплое выполнение — 20 мс. Каково среднее время выполнения? Сколько составит при полном устранении холодных стартов?

Интегральные и комбинаторные оценки

Задача 36. Мультипликативный эффект оптимизаций.

Оптимизация A увеличивает RPS в 1.6 раза, B — в 2.0 раза, C — в 1.25 раза. Исходный RPS — 150. Каков итоговый RPS после всех трёх оптимизаций?

Задача 37. Выбор оптимальной комбинации.

Цель — сократить LCP на 3.5 с. Доступны оптимизации: D (сокращение 2.0 с, 4 часа), E (1.5 с, 2 часа), F (1.2 с, 3 часа), G (0.8 с, 1 час). Найдите комбинацию с минимальными трудозатратами.

СВОДНЫЙ ПЕРЕЧЕНЬ ФОРМУЛ ДЛЯ РЕШЕНИЯ ПРАКТИЧЕСКИХ ЗАДАЧ

Настоящий перечень агрегирует и систематизирует всю совокупность математических зависимостей и формальных соотношений, задействованных в процессе решения практических задач, распределённых по всем пяти главам пособия. Данный компедиум может выполнять функцию компактного, структурированного справочного материала при проведении лабораторных практикумов, подготовке к промежуточной и итоговой аттестации, а также при выполнении оперативных инженерных оценок в рамках реальных производственных проектов, требующих быстрого, но математически обоснованного принятия архитектурных решений.

Раздел 1. Экономика производительности и системный анализ

Формула 1.1. Прирост конверсии от сокращения задержки

$$\Delta C = C_0 \cdot k \cdot \Delta T$$

Символ	Наименование	Единица измерения
ΔC	Абсолютный прирост коэффициента конверсии	процентные пункты (п.п.)
C_0	Исходный коэффициент конверсии	%
k	Коэффициент потерь конверсии на секунду задержки	1/с (эмпирически k≈0.07k≈0.07)
ΔT	Сокращение времени загрузки	секунды (с)

Область применения: Задачи 1.1, 1.2. Оценка экономического эффекта от мероприятий по ускорению загрузки страницы. Формула основана на эмпирически установленной линейной зависимости потерь конверсии от времени задержки в диапазоне 1–6 секунд.

Формула 1.2. Прирост выручки от увеличения конверсии

$$\Delta R = (C_{new} - C_{old}) \cdot AOV \cdot N_{sessions}$$

Символ	Наименование	Единица измерения
ΔR	Прирост выручки за период	руб.
C_{new}	Коэффициент конверсии после оптимизации	% (в долях единицы)
C_{old}	Коэффициент конверсии до оптимизации	% (в долях единицы)
AOV	Средний чек (Average Order Value)	руб.
$N_{sessions}$	Количество целевых сессий за период	шт.

Область применения: Задачи 1.1, 1.2. Перевод технического улучшения в финансовый результат.

Формула 1.3. Срок окупаемости инвестиций в оптимизацию

$$T_{payback} = \frac{Cost_{opt}}{\Delta R_{monthly}}$$

Символ	Наименование	Единица измерения
$T_{payback}$	Срок окупаемости	месяцы
$Cost_{opt}$	Совокупные затраты на оптимизацию	руб.
$\Delta R_{monthly}$	Месячный прирост выручки	руб./мес.

Область применения: Задача 1.2. Обоснование экономической целесообразности оптимизационных мероприятий.

Формула 1.4. Возврат на инвестиции (ROI) от оптимизации

$$ROI_{perf} = \frac{\left((C_{new} - C_{old}) \cdot AOV \cdot T_{sessions} - Cost_{opt}\right)}{Cost_{opt}}$$

Область применения: Глава 1, параграф 1.3. Стратегическое обоснование инвестиций в производительность.

Формула 1.5. Вероятность отказа пользователя от задержки (экспоненциальная модель)

$$P_{bounce}(T) = 1 - e^{-\alpha \cdot T}$$

Символ	Наименование	Единица измерения
P_{bounce}	Вероятность отказа (показатель отказов)	безразмерная (0…1)
α	Коэффициент чувствительности пользователя к задержке	1/с
T	Время загрузки страницы	секунды (с)

Область применения: Задача 1.6. Моделирование нелинейной зависимости пользовательского поведения от времени загрузки. При α=0.3 и T=5 с вероятность отказа составляет около 78%

Формула 1.6. Коэффициент загрузки системы (Traffic Intensity)

$$\rho = \frac{\lambda}{\mu}$$

Символ	Наименование	Единица измерения
ρ	Коэффициент загрузки (Traffic Intensity)	безразмерная (0…1)
λ	Интенсивность входящего потока заявок	заявок/с
μ	Интенсивность обслуживания	заявок/с

Область применения: Задачи 1.7, 2.2, 5.4. Ключевой показатель стабильности системы массового обслуживания. При ρ→1 время ответа асимптотически стремится к бесконечности.

Формула 1.7. Интенсивность обслуживания через среднее время обработки

$$\mu = \frac{1}{T_{service}}$$

Символ	Наименование	Единица измерения
μ	Интенсивность обслуживания	заявок/с
$T_{service}$	Среднее время обслуживания одной заявки	секунды (с)

Область применения: Задачи 1.7, 2.2. Связь между временны́ми и частотными характеристиками сервера.

Формула 1.8. Среднее время ответа в модели M/M/1

$$T_{sys} = \frac{\frac{1}{\mu}}{1 - \rho} = \frac{1}{\mu - \lambda}$$

Символ	Наименование	Единица измерения
T_{sys}	Среднее время пребывания заявки в системе	секунды (с)
μ	Интенсивность обслуживания	заявок/с
λ	Интенсивность входящего потока	заявок/с
ρ	Коэффициент загрузки	безразмерная

Область применения: Задачи 1.7, 2.2, 2.3. Демонстрирует нелинейный рост задержки при приближении к насыщению.

Формула 1.9. Среднее время ответа с кэшированием (средневзвешенное)

$$T_{avg} = p_{hit} \cdot T_{hit} + (1 - p_{hit}) \cdot T_{miss}$$

Символ	Наименование	Единица измерения
T_{avg}	Среднее время обработки запроса	мс
p_{hit}	Вероятность попадания в кэш (hit-ratio)	безразмерная (0…1)
T_{hit}	Время ответа при попадании в кэш	мс
T_{miss}	Время ответа при промахе мимо кэша	мс

Область применения: Задачи 1.8, 4.5. Оценка эффективности внедрения кэширующего слоя.

Формула 1.10. Ускорение от кэширования

$$Speedup = \frac{T_{without\ cache}}{T_{with\ cache}}$$

Область применения: Задачи 1.8, 4.5. Количественная мера эффекта от внедрения кэша.

Формула 1.11. Время последовательной загрузки при HTTP/1.1

$$T_{total} = \sum_{i=1}^{n} (RTT_i + \frac{S_i}{B})$$

Символ	Наименование	Единица измерения
T_{total}	Общее время загрузки	секунды (с)
RTT_i	Round-Trip Time для i-го ресурса	секунды (с)
S_i	Размер i-го ресурса	биты

B	Пропускная способность канала	бит/с	

Область применения: Задача 1.10. Упрощённая модель для сравнительного анализа протоколов.

Раздел 2. Сетевые ограничения и теория массового обслуживания

Формула 2.1. Время распространения сигнала в оптоволокне

$$T_{prop} = \frac{D}{v} = \frac{D * n}{c}$$

Символ	Наименование	Единица измерения
T_{prop}	Время распространения сигнала в одну сторону	секунды (с)
D	Физическая протяженность канала	метры (м)
v	Скорость света в среде передачи	м/с (метр/сек)
c	Скорость света в вакууме ($3 * 10^8$ м/с)	м/с (метр/сек)
n	Показатель преломления сердцевины волокна ($n \approx 1.48$)	безразмерная

Область применения: Задача 2.1. Фундаментальное физическое ограничение, не устранимое программными методами.

Формула 2.2. Round-Trip Time (RTT)

$$RTT = 2 * T_{prop} + \sum T_{routing}$$

Символ	Наименование	Единица измерения
T_{prop}	Время распространения сигнала в одну сторону	секунды (с)
RTT	Полное время кругового обхода	секунды (с)
$\sum T_{routing}$	Суммарные задержки на маршрутизаторах	секунды (с)

Область применения: Задача 2.1. При расчёте минимального RTT задержками на маршрутизаторах пренебрегают.

Формула 2.3. Bandwidth-Delay Product (BDP)

$$BDP = B * RTT$$

Символ	Наименование	Единица измерения
BDP	Произведение полосы пропускания на задержку	биты (бит)
RTT	Полное время кругового обхода (задержка)	секунды (с)
B	Пропускная способность канала	бит/сек

Область применения: Задача 2.5. Определяет минимальный объём данных «в полёте» для полной утилизации канала. Является ориентиром для настройки размера TCP-окна.

Формула 2.4. Закон Литтла

$$L = \lambda * W$$

Символ	Наименование	Единица измерения
L	Среднее число заявок в системе	штуки
λ	Интенсивность входящего потока	заявок/с
W	Среднее время пребывания заявки в системе	секунды (с)

Область применения: Задача 2.4. Инвариант для любых стационарных систем массового обслуживания. Используется для косвенной верификации измерений.

Формула 2.5. Совокупная задержка LCP

$$LCP = TTFB + T_{load} + T_{render}$$

Символ	Наименование	Единица измерения
LCP	Largest Contentful Paint	мс
$TTFB$	Time to First Byte	мс
T_{load}	Время загрузки LCP-элемента	мс
T_{render}	Время рендеринга LCP-элемента	мс

Область применения: Задача 2.6. Декомпозиция метрики для выявления доминирующей компоненты.

Формула 2.6. Cumulative Layout Shift (CLS) для единичного эпизода

$$CLS_{episode} = \frac{S_{impact}}{S_{viewport}} * \frac{D_{shift}}{H_{viewport}}$$

Символ	Наименование	Единица измерения
$CLS_{episode}$	Значение CLS для одного эпизода сдвига	безразмерная (0…1)
S_{impact}	Площадь области, затронутой сдвигом	пиксели^2
$S_{viewport}$	Общая площадь viewport	пиксели^2
D_{shift}	Расстояние сдвига элемента	пиксели
$H_{viewport}$	Высота viewport	пиксели

Область применения: Задача 2.7. Для упрощённого двумерного случая площадь заменяется высотой элемента, а расстояние сдвига — смещением по вертикали.

Формула 2.7. Позиция процентиля методом ближайшего ранга

$$i = \lceil P * n \rceil$$

Символ	Наименование	Единица измерения
i	Порядковый номер значения в отсортированной выборке	безразмерная
P	Целевой процентиль (в долях единицы)	безразмерная (0…1)
n	Объём выборки	штуки
$\lceil * \rceil$	Операция округления вверх до ближайшего целого	—

Формула 2.8. Проверка соблюдения SLO по процентилю

$$SLO_{violated} \Leftrightarrow \left(\frac{N_{exceed}}{N_{total}}\right) > (1 - P_{target})$$

Символ	Наименование	Единица измерения
N_{exceed}	Количество измерений, превысивших порог	штуки
N_{total}	Общее количество измерений	штуки
P_{target}	Целевой процентиль SLO (в долях единицы)	безразмерная

Область применения: Задача 2.9. Для SLO «P95 < 3.0 с» допустимая доля превышений составляет не более 1−0.95=0.05 (5%).

Формула 2.9. Общий коэффициент сжатия (мультипликативный)

$$k_{total} = k_1 * k_2 * \ldots * k_n = \prod_{i=1}^{n} k_i$$

Символ	Наименование	Единица измерения
k_i	Коэффициент сжатия на i-м этапе	безразмерная (0…1)
k_{total}	Общий коэффициент сжатия	безразмерная (0…1)

Область применения: Задачи 2.10, 5.9. Коэффициенты каждого этапа (минификация, Brotli) перемножаются, так как применяются последовательно к результату предыдущего этапа.

Раздел 3. Клиентская оптимизация

Формула 3.1. Время загрузки пропорционально размеру ресурса

$$T_{load} = \frac{S}{B} + RTT$$

Символ	Наименование	Единица измерения
T_{load}	Время загрузки ресурса	секунды (с)
S	Размер ресурса	биты
B	Пропускная способность	бит/с
RTT	Задержка кругового обхода	секунды (с)

Область применения: Задачи 3.1, 3.3, 3.4. Упрощённая модель для сравнительных оценок.

Формула 3.2. Средневзвешенный размер адаптивного изображения

$$S_{avg} = \sum_{i=1}^{k} p_i * S_i$$

Символ	Наименование	Единица измерения
S_{avg}	Средний размер загружаемого изображения	КБ
p_i	Доля трафика i-й категории устройств	безразмерная (0…1)
S_i	Размер изображения для i-й категории	КБ
k	Количество категорий устройств	штуки

Область применения: Задача 3.3. $\sum p_i = 1$

Формула 3.3. Экономия трафика

$$\Delta V = (S_{old} - S_{new}) * N_{impressions}$$

Символ	Наименование	Единица измерения
ΔV	Объём сэкономленного трафика	КБ, МБ, ГБ
S_{old}	Размер файла до оптимизации	КБ
S_{new}	Размер файла после оптимизации	КБ
$N_{impressions}$	Количество показов (загрузок)	штуки

Область применения: Задачи 3.2, 3.7.

Формула 3.4. Hit-ratio кэша

$$HitRatio = \frac{N_{cache\ hits}}{N_{total\ requests}}$$

Символ	Наименование	Единица измерения
$HitRatio$	Доля запросов, обслуженных из кэша	безразмерная (0…1)
$N_{cache\ hits}$	Количество попаданий в кэш	штуки
$N_{total\ requests}$	Общее количество запросов	штуки

Область применения: Задача 3.5. Основной показатель эффективности кэширования.

Формула 3.5. Время рендеринга при виртуализации (пропорциональная модель)

$$T_{virtual} = T_{full} * N_{visuable}/N_{total}$$

Символ	Наименование	Единица измерения
$T_{virtual}$	Время рендеринга с виртуализацией	мс
T_{full}	Время рендеринга без виртуализации	мс
$N_{visuable}$	Количество видимых элементов	штуки
N_{total}	Общее количество элементов	штуки

Область применения: Задачи 3.9, 5.2. Предполагает линейную зависимость времени рендеринга от количества DOM-узлов.

Формула 3.6. Загрузка CPU в абсолютных единицах

$$CPU_{load} = T_{reneders} * T_{per\ render}$$

Символ	Наименование	Единица измерения
CPU_{load}	Суммарное процессорное время на рендеринг	мс
$T_{reneders}$	Количество ререндеров в секунду	1/с
$T_{per\ render}$	Время одного ререндера	мс

Область применения: Задача 3.8. Позволяет оценить долю процессорного времени, затрачиваемого на избыточные операции.

Формула 3.7. Относительное снижение показателя

$$\Delta\% = \frac{X_{old} - X_{new}}{X_{old}} * 100\%$$

Область применения: Универсальная формула, используемая во всех задачах для оценки эффекта оптимизации.

Раздел 4. Серверная оптимизация

Формула 4.1. Время последовательного сканирования таблицы (Seq Scan)

$$T_{seq} = \frac{N_{rows}}{V_{scan}}$$

Символ	Наименование	Единица измерения
T_{seq}	Время полного сканирования таблицы	секунды (с)
N_{rows}	Общее количество строк в таблице	штуки
V_{scan}	Скорость сканирования (строк/с)	строк/с

Область применения: Задача 4.1. Оценка времени выполнения запроса без индекса.

Формула 4.2. Время поиска по индексу (Index Scan)

$$T_{index} = \frac{N_{matching}}{V_{index}}$$

Символ	Наименование	Единица измерения
T_{index}	Время поиска по индексу	секунды (с)
$N_{matching}$	Количество строк, удовлетворяющих условию	штуки

V_{index}	Скорость обработки при индексном доступе	строк/с

Область применения: Задача 4.1. Скорость индексного доступа на порядки выше скорости последовательного сканирования.

Формула 4.3. Баланс мощности кластера при масштабировании

$$N_{old} * U_{old} = N_{new} * U_{new}$$

Символ	Наименование	Единица измерения
N_{old}	Исходное количество серверов	штуки
U_{old}	Исходная средняя утилизация	% (в долях единицы)
N_{new}	Новое количество серверов	штуки
U_{new}	Новая средняя утилизация	% (в долях единицы)

Область применения: Задачи 1.5, 4.2. Следствие пропорциональности нагрузки и утилизации при неизменном профиле запросов.

Формула 4.4. Доля ключей, перераспределяемых при Consistent Hashing

$$f_{redist} = \frac{V_{new}}{V_{total\ new}}$$

Символ	Наименование	Единица измерения
f_{redist}	Доля перераспределяемых ключей	безразмерная (0…1)
V_{new}	Количество vnodes, добавляемых новым узлом	штуки
$V_{total\ new}$	Общее количество vnodes после масштабирования	штуки

Область применения: Задача 4.3. Теоретический минимум перераспределения данных при добавлении узла.

Формула 4.5. Время синхронной цепочки вызовов

$$T_{sync} = \sum_{i=1}^{m} (T_{proc,i} + T_{net,i})$$

Символ	Наименование	Единица измерения
T_{sync}	Общее время синхронного ответа	мс
$T_{proc,i}$	Время обработки в i-м сервисе	мс

$T_{net,i}$	Сетевая задержка до i-го сервиса	мс
m	Количество последовательно вызываемых сервисов	штуки

Область применения: Задача 4.4. Иллюстрация накопления задержек при синхронных вызовах в микросервисной архитектуре.

Формула 4.6. Общий hit-ratio многоуровневого кэша

$$HR_{total} = HR_1 + (1 = HR_1) * HR_2$$

Символ	Наименование	Единица измерения
HR_{total}	Общий hit-ratio системы кэширования	безразмерная (0…1)
HR_1	Hit-ratio первого уровня (L1)	безразмерная (0…1)
HR_2	Hit-ratio второго уровня (L2) на промахах L1	безразмерная (0…1)

Область применения: Задача 5.5. Обобщается на произвольное количество уровней.

Формула 4.7. Время сборки мусора пропорционально размеру кучи

$$T_{GC\ new} = T_{GC\ old} * \frac{H_{new}}{H_{old}}$$

Символ	Наименование	Единица измерения
HR_{total}	Общий hit-ratio системы кэширования	безразмерная (0…1)
HR_1	Hit-ratio первого уровня (L1)	безразмерная (0…1)
HR_2	Hit-ratio второго уровня (L2) на промахах L1	безразмерная (0…1)

Область применения: Задача 5.6. Упрощённая модель для Major GC в V8.

Формула 4.8. Время загрузки при параллельных соединениях HTTP/1.1

$$T_{1.1} = \left\lceil \frac{N_{resources}}{N_{connections}} \right\rceil * (RTT + T_{transfer})$$

Символ	Наименование	Единица измерения
$T_{1.1}$	Общее время загрузки при HTTP/1.1	безразмерная (0…1)
$N_{resources}$	Количество ресурсов	безразмерная (0…1)
$N_{connections}$	Максимальное число параллельных соединений (обычно 6)	безразмерная (0…1)

$\lceil * \rceil$	Округление вверх до целого	—

Область применения: Задачи 4.6, 5.7. Сравнительный анализ протоколов.

Формула 4.9. Время загрузки при мультиплексировании HTTP/2

$$T_2 = RTT + \max_{i=1\ldots n}(T_{transfer,i})$$

Область применения: Задачи 4.6, 5.7. При мультиплексировании все ресурсы передаются параллельно, общее время определяется самым «тяжёлым» из них.

Формула 4.10. Время установки TLS-соединения

$$T_{TLS} = N_{RTT} * RTT$$

Символ	Наименование	Единица измерения
T_{TLS}	Время TLS-рукопожатия	мс
N_{RTT}	Количество круговых обходов для протокола	штуки
RTT	Задержка одного кругового обхода	мс

Область применения: Задача 4.9. Для TLS 1.2 $N_{RTT} = 2$, TLS 1.3 $N_{RTT} = 1$, для TLS 1.3 0-RTT $N_{RTT} = 0$.

Раздел 5. Интегральные и комбинаторные оценки

Формула 5.1. Произведение коэффициентов улучшения

$$RPS_{new} = RPS_{old} * k_1 * k_2 * \ldots * k_m$$

Символ	Наименование	Единица измерения
RPS_{new}	Пропускная способность после оптимизации	запросов/с
RPS_{old}	Исходная пропускная способность	запросов/с
k_i	Коэффициент улучшения от i-го мероприятия	безразмерная ($k_i > 1$)

Область применения: Задача 5.3. Приближённая мультипликативная модель для независимых оптимизаций.

Формула 5.2. Общая загрузка CPU сервера (нормированная мощность)

$$C_{capasity} = N_{servers} * U_{avg}$$

Символ	Наименование	Единица измерения
$C_{capasity}$	Суммарная нормированная мощность кластера	безразмерные «единицы мощности»
$N_{servers}$	Количество серверов	штуки
U_{avg}	Средняя утилизация	% (в долях единицы)

Область применения: Задачи 1.5, 4.2. Используется для расчёта потребного количества узлов при изменении нагрузки или эффективности кода.

Формула 5.3. Комбинаторный перебор вариантов оптимизации

$$\forall S \in 2^{\{A,B,C\}}: \sum_{x \in S} \Delta T_x \geq \Delta T_{target}$$

Символ	Наименование	Единица измерения
S	Подмножество выбранных оптимизаций	—
ΔT_x	Сокращение задержки от оптимизации x	секунды (с)
ΔT_{target}	Целевое сокращение задержки	секунды (с)

Область применения: Задача 5.10. Формализация задачи выбора оптимальной комбинации мероприятий при ограниченных ресурсах.

ОТВЕТЫ К ЗАДАНИЯМ

<table>
<tr><th>№</th><th>Искомый показатель</th><th>Ответ</th></tr>
<tr><td>1.1</td><td>Прирост ежемесячной выручки</td><td>1 270 080 руб.</td></tr>
<tr><td>1.2</td><td>Срок окупаемости</td><td>≈ 3.9 месяца</td></tr>
<tr><td>1.3</td><td>а) Новое время ответа
б) Новое узкое место
в) Улучшение</td><td>130 мс
Компонент C (60 мс)
40.9%</td></tr>
<tr><td>1.4</td><td>Оптимальная последовательность</td><td>P3 → P1 → P2</td></tr>
<tr><td>1.5</td><td>а) Выводимые узлы
б) Годовая экономия</td><td>2 узла
288 000 руб.</td></tr>
<tr><td>1.6</td><td>а) P(2 с) и P(6 с)
б) Снижение P(bounce)</td><td>≈ 0.451; ≈ 0.835
≈ 32.6 п.п.</td></tr>
<tr><td>1.7</td><td>а) Коэффициент загрузки ρ
б) Рост среднего времени ответа</td><td>0.75
в 2.33 раза</td></tr>
<tr><td>1.8</td><td>Среднее время и ускорение</td><td>21.5 мс; в 3.72 раза</td></tr>
<tr><td>1.9</td><td>Увеличение FCP</td><td>на 230 мс</td></tr>
<tr><td>1.10</td><td>Выигрыш от HTTP/2</td><td>430 мс</td></tr>
<tr><td>2.1</td><td>Минимальный RTT</td><td>63.15 мс</td></tr>
<tr><td>2.2</td><td>а) μ
б) ρ
в) T_{sys}
г) L</td><td>66.67 заявок/с
0.75
60 мс
3 заявки</td></tr>
<tr><td>2.3</td><td colspan="2">
<table>
<tr><td>λ, заяв/с</td><td>10</td><td>30</td><td>50</td><td>70</td><td>80</td><td>90</td><td>95</td><td>99</td></tr>
<tr><td>$Tsys$, м</td><td>11.1</td><td>14.3</td><td>20.0</td><td>33.3</td><td>50</td><td>100.0</td><td>200.0</td><td>1000.0</td></tr>
</table>
Комментарий: При увеличении λ с 90 до 99 заявок/с (рост на 10%) время ответа возрастает в 10 раз, что иллюстрирует катастрофическую нелинейность вблизи насыщения.
</td></tr>
<tr><td>2.4</td><td>Фактическая интенсивность потока запросов λ; соответствует ли она номинальной нагрузке?</td><td>266.7 заявок/с; не соответствует</td></tr>
<tr><td>2.5</td><td>а) Bandwidth-Delay Product в мегабитах и мегабайтах;
б) минимальный размер TCP-окна для полной утилизации канала</td><td>3 Мбит; 0.375 МБ;
0.375 МБ</td></tr>
<tr><td>2.6</td><td>Относительное улучшение LCP после сокращения TTFB</td><td>30%</td></tr>
<tr><td>2.7</td><td>Значение Cumulative Layout Shift (CLS) для единичного инцидента</td><td>0.028</td></tr>
<tr><td>2.8</td><td>а) Среднее арифметическое;
б) Медиана (P50);
в) P75;
г) P95;
д) P99.</td><td>а) 148.5 мс;
б) 77.5 мс;
в) 100 мс;
г) 350 мс;
д) 1200 мс</td></tr>
<tr><td>2.9.</td><td>Находится ли система в рамках SLO?</td><td>Нет, SLO нарушено</td></tr>
<tr><td>2.10</td><td>а) Итоговый размер файла;
б) общий коэффициент сжатия относительно исходного</td><td>70.4 КБ;
0.147 (сжатие в 6.8 раза)</td></tr>
<tr><td>3.1</td><td>Сокращение FCP</td><td>≈ 344 мс</td></tr>
<tr><td>3.2</td><td>а) Размеры файлов
б) Экономия на 10 000 показов</td><td>WebP: 595 КБ; AVIF: 425 КБ</td></tr>
</table>

		WebP: 2.55 ГБ; AVIF: 4.25 ГБ
3.3	Средний размер без и с адаптацией	450 КБ; 283.25 КБ
3.4	а) Размеры после сжатия б) Превосходство Brotli	Gzip: 360 КБ; Brotli: 264 КБ; на 26.7%
3.5	а) Hit-ratio б) Доля полных загрузок	80% 10%
3.6	Средний объём кода на сессию	530 КБ
3.7	Экономия трафика	3.84 МБ
3.8	а) Загрузка CPU до и после б) Снижение	420 мс/с; 52.5 мс/с 87.5%
3.9	Время рендеринга с виртуализацией	≈ 1.28 мс
3.10	Сокращение времени до читаемого текста	600 мс
4.1	а) Время без индекса б) Время с индексом в) Ускорение	25 с 0.25 мс в 100 000 раз
4.2	а) Необходимо добавить б) Новая утилизация	3 сервера ≈ 98%
4.3	а) Количество vnodes б) Доля перераспределяемых ключей	600 и 750 20%
4.4	а) Синхронное время б) Время с асинхронным C	165 мс 90 мс
4.5	а) Среднее до б) Среднее после в) Ускорение	10.2 мс 3.2 мс в 3.19 раза
4.6	а) Время HTTP/1.1 б) Время HTTP/2	300 мс 50 мс
4.7	а) Размеры ответов б) Экономия	JSON: 357 КБ; Protobuf: 153 КБ; 204 КБ (57.1%)
4.8	а) Время без DataLoader б) Время с DataLoader в) Ускорение	328 мс 24 мс в 13.7 раза
4.9	Время соединения для 4 сценариев	80, 40, 40, 0 мс
4.10	а) Суммарное время б) Среднее время в) Сокращение	59 000 мс 59 мс на 38.1%
5.1	Вклад Code Splitting в улучшение LCP	≈ 3 с (из 3 с улучшения)
5.2	TBT после виртуализации и улучшение	≈ 3.28 мс; на 99.6%
5.3	Расчётный RPS через коэффициенты	600 RPS (близко к 610)
5.4	а) Порог нелинейности б) ρ на пороге	300 RPS 0.5
5.5	а) Общий hit-ratio б) Доля запросов к БД	94% 6%
5.6	а) Новое время паузы GC б) Доля времени GC	40 мс 1.5% → 0.5%
5.7	а) Время HTTP/1.1 б) Время HTTP/2	≈ 560 мс 80 мс
5.8	Превышение и минимальное сокращение	20%; 70 КБ

5.9	а) Коэффициент минификации б) Коэффициент Brotli в) Общий коэффициент	0.6 0.219 0.131
5.10	а) Комбинации для цели б) Минимальные трудозатраты	A+B, A+B+C, B+C A+B (9 часов)
1	$\Delta C=C_0 \cdot k \cdot \Delta T$, откуда $\Delta T=\Delta C/(C_0 \cdot k)=$ $(0.041-0.034)/(0.041\times0.07)=0.007/0.00287 \approx$ $2.44\Delta T=\Delta C/(C_0 \cdot k)=(0.041-0.034)/(0.041\times0.07)$ $=0.007/0.00287\approx2.44$ с = 2440 мс.	2440 мс.
2	$\Delta C=2.8\%\times0.07\times1.5=0.294$ п.п. Месячный прирост: $0.00294\times5100\times60000=899640$ руб. Максимальный бюджет: $Cost_{op}t=\Delta R_{monthly}\times T_{payback}=899\,640\times6 = 5397840$ руб.	5397840 руб.
3	$\Delta C=\Delta R/(AOV \cdot N_{sessions}) = 756000/(4200\times100000)$ $= 0.0018=0.18$ п.п. По формуле $\Delta C=C_0 \cdot k \cdot \Delta T$, проверяем: $3.0\%\times0.07\times1.8=0.378$ п.п. Не совпадает, значит, исходная конверсия не 3.0%. Решаем обратно: $C_0=\Delta C/(k \cdot \Delta T)=0.0018/(0.07\times1.8)=0.0018/0.126\approx0.0143=1.43$ % . Исходная конверсия — 1.43%.	1.43%.
4	а) Новые времена: A=25, B=20, C=55, D=40. Узкое место — C (55 мс). Общее время = 140 мс. б) Целевое общее время — 100 мс, сумма неснижаемых компонентов = 25+20+40 = 85 мс. Допустимое время C = 100 − 85 = 15 мс. Требуемое ускорение C: 55/15≈3.6755/15≈3.67 раза, снижение на (55−15)/55×100%≈72.7%.	Общее время = 140 мс. Допустимое время = 15 мс. Требуемое ускорение ≈72.7%.
5	$P(4)=1-e^{-1.6} \approx0.798$. $P(2)=1-e^{-0.8}\approx0.551$. Доля уходящих при 4 с, которые остались бы при 2 с: $(0.798-0.551)/0.798\times100\% \approx 31.0\%$. Вернуть можно 31% от ушедших.	31%
6	$T_{prop,MSK}=1600000\times1.48/(3\times10^8)\approx7.89$ мс, $RTT_{MSK}\approx15.8$ $T_{prop,FRA}=3\,800\,000\times1.48/(3\times10^8)\approx18.7$ $RTT_{FRA}\approx37.5$мс. Разница: 37.5−15.8≈21.7 мс.	21.7 мс.
7	а) $\mu=1/0.006\approx166.7$ заявок/с. $\rho=120/166.7=0.72$. $T_{sys}= =(1/166.7)/(1-0.72)=0.006/0.28\approx21.4$ мс. Это время внутри сервера. RTT добавляет 2×40=80 мс. Итого модель предсказывает 21.4+80≈101.4 мс, что существенно меньше наблюдаемых 320 мс. б) Модель M/M/1 не соответствует — вероятно, есть дополнительные источники задержки (внешние вызовы, дисковые операции).	а) 320 мс. б) не соответствует
8	$\lambda=L/W=8/0.040=200$ заявок/с. $\mu=\lambda/\rho=200/0.8=250$ заявок/с	200 заявок/с. 250 заявок/с
9	$T_{sys}=1/(\mu-\lambda)$. Требуем $T_{sys}>0.050$ с. $\Rightarrow300-\lambda<20\Rightarrow\lambda>280$ $1/(300-\lambda)>0.050\Rightarrow300-\lambda<20\Rightarrow\lambda>280$ заявок/с.	280 заявок/с.
10	$BDP=15\times10^6\times0.6=9\times10^6$ бит = 9 Мбит = 1.125 МБ.	1.125 МБ.
11	Сервер A: $\rho_A=300/500=0.6$, $T_A=(1/500)/(1-0.6)=0.002/0.4=5$ $T_A=(1/500)/(1-0.6)=0.002/0.4=5$ мс. Сервер B: на каждую очередь $\lambda_B=150$, $\rho_B=150/250=0.6$ $T_B=(1/250)/(1-0.6)=0.004/0.4=10$ мс. Сервер A эффективнее в 2 раза.	Сервер A эффективнее в 2 раза.
12	$\lambda_{real}=L/W=4.2/0.0014=3000$ заявок/с. Расхождение: $(3500-3000)/3500\times100\%\approx14.3\%$. Да, расхождение значительное — возможно, часть запросов обслуживается из локального кэша, не доходя до Redis.	Расхождение значительное

13	Эпизод 1: impact fraction = 200/1000 = 0.2, distance fraction = 200/1000 = 0.2, CLS_1 = 0.04. Эпизод 2: impact fraction = 120/1000 = 0.12, distance fraction = 120/1000 = 0.12, CLS_2 = 0.0144. Итоговый CLS (без учёта кластеризации) = 0.04 + 0.0144 = 0.0544.	0.0544
14	Допустимая доля превышений для P99 = 1%. Фактическая доля = 300 / 50 000 = 0.006 = 0.6%. 0.6%<1%, следовательно, SLO выполняется.	SLO выполняется
15	$k_{brotli\_vs\_gzip}$ =95/140≈0.679 (Brotli на 32.1% эффективнее Gzip). k_{total} =95/600≈0.158 (сжатие в 6.3 раза).	Brotli на 32.1% эффективнее Gzip; сжатие в 6.3 раза
16	Время загрузки пропорционально размеру. Некритическая часть = 210 КБ, время её загрузки = 600×(210/240)=525 мс. Однако FCP теперь определяется инлайном — задержка сокращается на все 600 мс.	600 мс.
17	WebP: 720×0.72=518.4 КБ, экономия = 720−518.4=201.6 КБ. AVIF: 720×0.52=374.4 КБ, экономия = 720−374.4=345.6 КБ. На 20 000 показов: AVIF экономит 345.6×20000=6912000 КБ ≈ 6.9 ГБ. WebP экономит 201.6×20000≈ 4.0 ГБ. Выбор — AVIF.	AVIF.
18	$Savg = 0.5 \times 80 + 0.3 \times 180 + 0.2 \times 400 = 40 + 54 + 80 = 174$КБ Без адаптации — 400 КБ. Экономия = 400−174=226 КБ, или 56.5%.	56.5%
19	Hit-ratio = 56000/80000=0.70 (70%). Доля с передачей тела = (16000+8000)/80000=0.30 (30%).	30%
20	Без lazy: 50×150=7500 КБ. С lazy: 12×150=1800 КБ. Экономия на сессию = 5700 КБ. На 1000 сессий = 5700000 КБ ≈ 5.7 ГБ.	5.7 ГБ
21	$T_{virtual}$=1200×(25/15000)=1200×0.00167=2.0 мс. Улучшение — на 99.8%.	99.8%
22	До: 200×2.5=500 мс/с (50% CPU). После: 25×2.5=62.5 мс/с (6.25% CPU). Снижение: (500−62.5)/500×100%=87.5%.	87.5%
23	Без оптимизации — 800 мс. С swap — 0 мс. Сокращение — 800 мс.	800 мс.
24	HTTP/1.1: ⌈48/6⌉=8 ресурсов на соединение. $T_{1.1}$ =8×(60+30)=720 мс. HTTP/2: T_2=60+max(30)=90 мс.	Ускорение в 8 раз.
25	k_{total} =180/1500=0.12. Сжатие в 8.33 раза.	Сжатие в 8.33 раза.
26	T_{seq} =10000000/150000≈66.7 с. T_{index} =800/2500000=0.00032 с. Ускорение = 66.7/0.00032≈208438 раз.	≈208438 раз.
27	Текущая мощность: 6×0.65=3.9 у.е. Новая требуемая мощность: 3.9×1.4=5.46 у.е. Требуется узлов: 5.46/0.65≈8.4→9 узлов. Добавить: 9−6=3 узла.	3 узла
28	До: 5×200=1000 vnodes. После: 6×200=1200 vnodes. Новый узел получает 200/1200=1/6≈16.7% ключей. Перераспределено 16.7%.	Перераспределено 16.7%
29	T_{sync}=25+20+35+15+50=145 мс. Если C сделать асинхронным, не входящим в критический путь: 25+20+35=80 мс.	145 мс, 80 мс.
30	Общий HR = 0.55+0.45×0.80=0.55+0.36=0.91 (91%). Доля в БД = 1−0.91=0.09 (9%).	9%.

31	T_{GC_new}=150×(200/600)=50 мс. Доля до: 150/10000=1.5%. После: 50/10000=0.5%.	до: 1.5%, после: 0.5%.
32	JSON: 1200×380=456000 байт = 456 КБ. Protobuf: 11200×160=192000 байт = 192 КБ. Экономия на запрос = 264 КБ. На 5000 запросов = 1320000 КБ ≈ 1.32 ГБ.	1.32 ГБ.
33	TLS 1.2 полное: 2×70=140 мс. TLS 1.3 полное: 1×70=70 мс. TLS 1.2 возобновление: 1×70=70 мс. TLS 1.3 0-RTT: 0 мс.	0 мс.
34	Без: (1+60)×10=610 мс. С: (1+2)×10=30 мс. Ускорение = 610/30≈20.3 раза.	20.3 раза.
35	Суммарное время: 120×400+680×20=48000+13600=61600 мс. Среднее: 61600/800=77 мс. Без холодных стартов: 800×20=16000 мс, среднее — 20 мс. Сокращение среднего в 3.85 раза.	Сокращение среднего в 3.85 раза.
36	RPS_{new} =150×1.6×2.0×1.25=150×4.0=600 RPS. Рост в 4 раза.	Рост в 4 раза.
37	Перебор: E+F+G = 1.5+1.2+0.8 = 3.5 с (2+3+1 = 6 ч). D+E = 2.0+1.5 = 3.5 с (4+2 = 6 ч). D+F+G = 2.0+1.2+0.8 = 4.0 с (4+3+1 = 8 ч). Минимальные трудозатраты — 6 часов, обе комбинации E+F+G и D+E. Выбираем D+E (меньше компонентов, проще реализация).	D+E

Примечание к задаче 2.8: В инженерной практике для расчёта процентилей часто используют линейную интерполяцию между соседними значениями. В целях простоты и однозначности в данном решении использован метод ближайшего ранга (nearest rank). Различие в результатах при использовании интерполированного метода для выборок малого объёма может составлять единицы миллисекунд и не влияет на качественные выводы.

СПИСОК ЛИТЕРАТУРЫ

1. Голдратт, Э.М. Цель: Процесс непрерывного совершенствования / Э.М. Голдратт, Дж. Кокс. — Минск: Попурри, 2019. — 400 с.
2. Клеппман, М. Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб.: Питер, 2018. — 640 с.
3. Администрирование локально-вычислительных сетей под управлением BaseALT Linux Server: учебно-методическое пособие / И. С. Поломошнов, О. А. Литвиненко. — Улан-Удэ: Издательство Бурятского госуниверситета им. Д. Банзарова, 2026. — 140 с. ISBN 978-5-9793-2138-7
4. Метод математической индукции и комбинаторика: учебное пособие / И. С. Поломошнов, О. А. Литвиненко. — Улан-Удэ: Издательство Бурятского госуниверситета им. Д. Банзарова, 2026. — 80 с.
5. Ньюмен, С. Создание микросервисов. — 2-е изд. — СПб.: Питер, 2023. — 624 с.
6. Таненбаум, Э. Компьютерные сети / Э. Таненбаум, Д. Уэзеролл. — 6-е изд. — СПб.: Питер, 2024. — 992 с.
7. Grigorik, I. High Performance Browser Networking. — O'Reilly Media, 2013. — 400 p.
8. Osmani, A. Image Optimization. — Smashing Magazine, 2021. — 528 p.
9. Langley, A., Riddoch, A., Wilk, A., et al. The QUIC Transport Protocol: Design and Internet-Scale Deployment // Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17). — ACM, 2017. — P. 183–196.
10. Walton, P. Web Performance 101. — CSS-Tricks, 2023. [Электронный ресурс].
11. MDN Web Docs. Critical Rendering Path. — Mozilla, 2024. [Электронный ресурс].
12. Web Almanac. Chapter 12: Performance. — HTTP Archive, 2024. [Электронный ресурс].

Приложение 1. Пример конфигурации Nginx для статики с HTTP/2 и Brotli

```
server {
    listen 443 ssl http2;
    server_name ilspo.ru;

    # Пути к SSL-сертификатам
    ssl_certificate     /etc/letsencrypt/live/ilspo.ru/certs/ilspo.crt;
    ssl_certificate_key /etc/letsencrypt/live/ilspo.ru/certs/ilspo.key;
    ssl_protocols TLSv1.3;

    root /var/www/app/dist;
    index index.html;

    # Включение Brotli-сжатия
    brotli on;
    brotli_comp_level 6;
    brotli_types text/plain text/css application/javascript application/json image/svg+xml;

    # Кэширование статических ресурсов с хешами
    location ~* \.[a-f0-9]{8}\.(js|css|png|jpg|webp|avif|svg)$ {
        expires 1y;
        add_header Cache-Control "public, immutable";
    }

    # Для index.html кэширование отключено
    location / {
        try_files $uri $uri/ /index.html;
        add_header Cache-Control "no-cache, must-revalidate";
    }
}
```

Приложение 2. Глоссарий терминов

А

Агрегация (Aggregation) — процесс сбора, приведения к единому формату и объединения разрозненных данных (метрик, логов, событий) из множества источников в единый структурированный набор для последующего анализа, визуализации или алертинга.

Адресное пространство — диапазон адресов оперативной памяти, выделенный конкретному процессу операционной системой. В контексте in-memory хранилищ — область памяти, в которой данные находятся в непосредственном доступе процесса без обращения к внешним сетевым или дисковым ресурсам.

Алерт (Alert) — автоматическое уведомление, генерируемое системой мониторинга при выходе контролируемой метрики за пределы заданного порогового значения. Служит триггером для реагирования дежурного персонала на инцидент.

Аппроксимация (Approximation) — замена одного математического объекта (функции, распределения, модели) другим, более простым или удобным для анализа, с сохранением существенных свойств исходного объекта в пределах допустимой погрешности.

Артефакт — любой файл или объект, полученный в результате работы программного инструмента (сборщика, компилятора, системы логирования), который может быть сохранён, передан или проанализирован. Примеры: собранный бандл, лог-файл, отчёт профилировщика.

Асинхронность — модель выполнения операций, при которой инициировавший операцию поток не блокируется в ожидании её завершения, а продолжает обработку других задач. Результат операции обрабатывается по факту его готовности через колбэк, Promise или событие.

Атомарность — свойство операции или транзакции, гарантирующее, что она будет выполнена либо полностью, либо не выполнена вовсе. Промежуточные, частично применённые состояния не видны другим конкурентным процессам.

Б

Балансировка нагрузки (Load Balancing) — техника распределения входящего сетевого трафика или запросов между пулом доступных серверных

узлов с целью оптимизации утилизации ресурсов, максимизации пропускной способности и минимизации времени отклика.

Балансировщик нагрузки (Load Balancer) — аппаратное или программное устройство, выступающее единой точкой входа для клиентских запросов и осуществляющее их диспетчеризацию на бэкенд-узлы по заданному алгоритму.

Базис — фундаментальная основа, совокупность принципов или исходных положений, на которых строится теория, методология или архитектурное решение.

Бандл (Bundle) — результирующий файл (или набор файлов), полученный в процессе сборки фронтенд-приложения, содержащий объединённый, минифицированный и, возможно, транспилированный исходный код и зависимости, предназначенный для загрузки браузером.

Батчевая транзакция (Batch Transaction) — операция, объединяющая множество однотипных атомарных действий (например, вставку или обновление записей) в одну групповую транзакцию для сокращения количества сетевых обращений и накладных расходов.

Браузер (Browser) — прикладное программное обеспечение, выступающее клиентским агентом пользователя для навигации, загрузки и отрисовки веб-страниц, исполнения скриптов и взаимодействия с веб-серверами.

Браузерный агент (User Agent) — строка или программный компонент, идентифицирующий браузер, его версию и операционную систему клиента перед сервером. В более широком смысле — сам браузер как активный участник клиент-серверного взаимодействия.

Буфер — временная область памяти, используемая для сглаживания разницы в скорости поступления и обработки данных между двумя компонентами системы, например, между сетью и диском или между продюсером и консюмером очереди.

В

Валидация (Validation) — процесс проверки соответствия данных заданным правилам, ограничениям или ожидаемому формату. В контексте кэширования — проверка актуальности закэшированной копии путём сверки с источником (например, через ETag или Last-Modified).

Вариативность (Variability) — степень разброса, неоднородности или изменчивости значений наблюдаемой метрики во времени или между различными экземплярами системы.

Вариант (Variant) — одна из возможных реализаций, конфигураций или версий системы, алгоритма или компонента, сравниваемая с другими в рамках анализа или A/B-тестирования.

Веб-ориентированная система (Web-based System) — программная система, доступ к которой осуществляется посредством веб-браузера по протоколам HTTP/HTTPS, а бизнес-логика распределена между клиентской и серверной частями.

Векторное изображение — изображение, описываемое математическими примитивами (линиями, кривыми, полигонами), а не растровой сеткой пикселей. Масштабируется без потери качества. Форматы: SVG.

Вертикальное масштабирование (Scaling Up) — увеличение производительности отдельного сервера за счёт наращивания его аппаратных ресурсов: добавления CPU, оперативной памяти, замены дисков на более производительные.

Виртуальная машина (Virtual Machine, VM) — программная эмуляция физического компьютера, функционирующая в изолированном окружении и имеющая собственную гостевую операционную систему, виртуальные аппаратные ресурсы и сетевой стек.

Высоконагруженный сервис (High-Load Service) — программный сервис, рассчитанный на обработку значительного количества конкурентных запросов (от тысяч до миллионов RPS), требующий специфических архитектурных решений по масштабированию, кэшированию и балансировке.

Вычислительное ядро (CPU Core) — физический или логический процессорный блок, способный независимо выполнять поток инструкций. Количество ядер определяет параллелизм обработки на уровне оборудования.

Г

Горизонтальное масштабирование (Scaling Out) — увеличение общей производительности системы за счёт добавления новых серверных узлов (инстансов), работающих параллельно, вместо наращивания мощности одного узла.

Д

Декодирование (Decoding) — процесс преобразования закодированных (сжатых, сериализованных) данных обратно в формат, пригодный для непосредственного использования приложением или отображения.

Декомпозиция кода (Code Decomposition) — разбиение монолитной кодовой базы на логически обособленные, слабо связанные модули, чанки или компоненты с целью упрощения разработки, тестирования и оптимизации загрузки.

Демпфер (Damper) — компонент или механизм, предназначенный для гашения колебаний, сглаживания пиков нагрузки или амортизации ударных воздействий на систему, например, очередь сообщений перед воркером.

Децентрализация (Decentralization) — архитектурный принцип, при котором управление и принятие решений распределены между множеством равноправных узлов, а не сосредоточены в едином центре.

Дихотомия (Dichotomy) — способ классификации, при котором множество объектов делится на два взаимоисключающих подмножества по определённому критерию (например, лабораторные vs полевые метрики).

Домен — **1.** В DNS: часть иерархического пространства имён, идентифицируемая уникальным доменным именем. **2.** В контексте виртуального почтового хостинга: административная единица, объединяющая почтовые ящики, алиасы и политики, относящиеся к одной организации.

З

Закон Литтла (Little's Law) — фундаментальный инвариант теории массового обслуживания: $L = \lambda \cdot W$, где L — среднее число заявок в системе, λ — средняя интенсивность входного потока, W — среднее время пребывания заявки в системе. Справедлив для любых стационарных систем.

И

Имманентность (Immanence) — свойство, внутренне присущее объекту или процессу, неотъемлемое от его природы. Например, имманентная задержка распространения сигнала в оптоволокне.

Имплементация (Implementation) — практическая реализация проектного решения, алгоритма или спецификации в виде работающего программного или аппаратного компонента.

Инвариант (Invariant) — утверждение или соотношение, остающееся неизменным при определённых преобразованиях или в процессе функционирования системы. Закон Литтла — пример инварианта для стационарных очередей.

Инвалидация (Invalidation) — процесс принудительного признания закэшированной копии данных недействительной, что заставляет систему при следующем обращении запросить актуальную версию из источника.

Индекс в реляционной СУБД (Database Index) — специализированная, физически материализованная структура данных (чаще всего B-tree), обеспечивающая ускоренный селективный доступ к кортежам таблицы по значениям индексируемых столбцов в обход полного сканирования (Full Table Scan).

Инлайнинг (Inlining) — техника встраивания содержимого внешнего ресурса (CSS, JavaScript, изображения в формате Base64) непосредственно в HTML-документ для сокращения количества сетевых запросов в критическом пути рендеринга.

Интерполяция (Interpolation) — метод нахождения промежуточных значений величины по имеющемуся дискретному набору известных значений. В контексте метрик — способ оценки значения процентиля между двумя соседними измерениями.

Интерпретация (Interpretation) — процесс наделения полученных количественных данных смыслом, выявления закономерностей и формулирования выводов в контексте решаемой задачи.

In-memory хранилище (In-Memory Data Store) — база данных или кэш, хранящая весь рабочий набор данных исключительно в оперативной памяти, что обеспечивает экстремально низкие задержки доступа (микро- и наносекунды) по сравнению с дисковыми хранилищами.

К

Каскадная таблица стилей (Cascading Style Sheets, CSS) — формальный язык описания внешнего вида документа, определяющий правила отрисовки элементов HTML (шрифты, цвета, отступы, позиционирование). Каскадность означает возможность комбинирования и наследования правил из разных источников.

Каскадное обновление (Cascade Update) — распространение изменений одной сущности на все зависимые или дочерние сущности в иерархии данных, могущее вызвать волну повторных рендерингов или модификаций в БД.

Каскадные вторичные блокировки — ситуация в критическом пути рендеринга, когда CSS блокирует выполнение JavaScript, а JavaScript, в свою очередь, блокирует парсер HTML, создавая цепочку взаимных задержек.

Квантификация (Quantification) — процесс присвоения измеряемому свойству или явлению численного значения, перевод качественных характеристик в количественную, математически обрабатываемую форму.

Кейс (Case) — описание конкретной проблемной ситуации, проекта или инцидента, используемое в обучении или анализе. Кейс-стади (Case Study) — метод исследования, предполагающий глубокое, всестороннее изучение отдельного кейса.

Когерентность данных (Data Coherence) — свойство распределённой системы, гарантирующее, что все её узлы в любой момент времени (или в пределах заданного окна) имеют непротиворечивое представление о состоянии данных.

Когнитивная разгрузка (Cognitive Offloading) — перенос части вычислительной или мнемонической нагрузки с пользователя на интерфейс или с разработчика на фреймворк (например, мемоизация избавляет разработчика от ручного отслеживания зависимостей пересчёта).

Колбэк (Callback) — функция, передаваемая в качестве аргумента другой функции и вызываемая асинхронно по завершении определённой операции или при наступлении события.

Конвертация (Conversion) — 1. В контексте веб-аналитики: совершение пользователем целевого действия (покупка, регистрация). 2. В контексте данных: преобразование из одного формата в другой (JSON в Protobuf, JPEG в WebP).

Контейнер (Container) — изолированное, легковесное окружение для запуска приложений, использующее ядро хостовой ОС, но имеющее собственную файловую систему, сетевой стек и лимиты ресурсов. В отличие от виртуальной машины, не эмулирует аппаратное обеспечение.

Корреляция (Correlation) — статистическая мера взаимосвязи между двумя или более переменными. Наличие корреляции не означает причинно-

следственной связи. Положительная корреляция: с ростом X растёт Y. Отрицательная: с ростом X Y убывает.

Кортеж входных аргументов (Input Argument Tuple) — упорядоченный, фиксированный набор значений, передаваемый функции в момент её вызова и однозначно определяющий (в случае чистой функции) результат её выполнения.

Куки (Cookies) — небольшие фрагменты данных, сохраняемые браузером на стороне клиента по инициативе веб-сервера и автоматически прикрепляемые к каждому последующему запросу к тому же домену. Используются для поддержания сессий, трекинга и персонализации.

Кэширование (Caching) — техника сохранения результатов часто запрашиваемых или дорогих для вычисления данных в быстром промежуточном хранилище (кэше) для ускорения повторных обращений.

Л

Латентность (Latency) — интервал времени между инициацией запроса и получением первого байта ответа. Обычно измеряется в миллисекундах (мс) и является ключевой метрикой производительности распределённых систем.

Лимит (Limit) — жёстко заданное граничное значение (количества соединений, потребляемой памяти, размера очереди и т.п.), превышение которого предотвращается системой принудительно — через отклонение запроса, троттлинг или ошибку.

М

Масштабирование (Scaling) — способность системы справляться с ростом рабочей нагрузки за счёт добавления ресурсов. См. также: вертикальное масштабирование, горизонтальное масштабирование.

Межсервисное взаимодействие (Inter-Service Communication) — обмен данными и вызов процедур между независимыми сервисами в распределённой системе, осуществляемый по сети с использованием определённого протокола (REST, gRPC, GraphQL, очереди сообщений).

Мемоизация (Memoization) — техника кэширования результатов выполнения функций с привязкой к их входным аргументам. При повторном вызове с теми же аргументами вычисление не производится, а возвращается сохранённый результат. Применяется для предотвращения избыточных ререндеров и дорогих вычислений.

Метрика (Metric) — количественно измеримый показатель, характеризующий определённое свойство системы, процесса или пользовательского опыта. Метрика должна быть наблюдаема, измерима и интерпретируема.

Микрофриз (Micro-freeze) — кратковременная (единицы-сотни миллисекунд) задержка отрисовки интерфейса, вызванная блокировкой основного потока выполнения (Main Thread) ресурсоёмкой операцией. Визуально воспринимается пользователем как «залипание» или прерывистость анимации.

Множество (Set) — абстрактная структура данных, хранящая уникальные, неупорядоченные элементы и поддерживающая операции добавления, удаления и проверки принадлежности. В Redis — тип данных Set.

Модель OSI (Open Systems Interconnection) — эталонная сетевая модель, декомпозирующая процесс сетевого взаимодействия на семь иерархических уровней: от физического (L1) до прикладного (L7).

Модель оценок M/M/1 — простейшая аналитическая модель системы массового обслуживания с одним обслуживающим прибором, пуассоновским входящим потоком (M), экспоненциальным временем обслуживания (M) и неограниченной очередью. Используется для грубых оценок пропускной способности и задержек.

Мониторинг (Monitoring) — непрерывный, автоматизированный процесс сбора, агрегации и визуализации телеметрических данных о состоянии системы с целью своевременного обнаружения отклонений и генерации алертов.

Мутирование данных (Data Mutation) — любая операция, изменяющая состояние существующей структуры данных (создание, обновление, удаление). Противопоставляется иммутабельному подходу, где изменения порождают новую копию.

Мультипликативность (Multiplicativity) — свойство эффекта, при котором совокупный результат от применения нескольких факторов равен произведению (а не сумме) их индивидуальных вкладов.

Н

Необходимое и достаточное условие — логическая связка: условие A необходимо для B, если без A невозможно B; условие A достаточно для B, если из наличия A всегда следует B.

Низконагруженный сервис (Low-Load Service) — сервис с небольшим объёмом трафика, к которому не предъявляются жёсткие требования по горизонтальному масштабированию и сверхнизкой латентности.

О

Окружение (Environment) — совокупность аппаратных, программных и сетевых ресурсов, в которых функционирует система. Типичные окружения: разработки (dev), тестирования (staging), продуктивное (production).

Оффлоад (Offload) — перенос выполнения задачи или функции с одного компонента или устройства на другой, как правило, для разгрузки основного процессора или критического пути. Пример: оффлоад TLS-терминации на балансировщик.

П

Пагинация (Pagination) — техника разбиения большого набора данных на дискретные страницы фиксированного размера, передаваемые клиенту по запросу, для предотвращения перегрузки сервера и сети.

Парадигма (Paradigm) — устоявшаяся модель, подход или стиль программирования/проектирования, определяющий способ структурирования кода и решения задач (ООП, функциональное, реактивное, событийно-ориентированное).

Паразитные циклы рендеринга (Parasitic Render Cycles) — избыточные, не обусловленные действительными изменениями данных повторные проходы механизма согласования (reconciliation) и отрисовки компонентов, потребляющие ресурсы CPU без видимой пользы.

Парсер (Parser) — компонент программного обеспечения, осуществляющий синтаксический анализ (парсинг) входной последовательности символов или токенов в соответствии с заданной грамматикой для построения структуры данных (например, DOM или AST).

Парсинг (Parsing) — процесс синтаксического анализа текста (HTML, CSS, JSON, кода программы), результатом которого является структурированное представление, пригодное для дальнейшей машинной обработки.

Паттерн (Pattern) — типовое, многократно проверенное на практике решение часто встречающейся проблемы проектирования или архитектуры в заданном контексте.

Перманентность (Permanence) — свойство состояния или процесса сохраняться непрерывно, без перерывов и остановок, на протяжении длительного периода времени.

Персистентные данные (Persistent Data) — данные, сохраняемые на энергонезависимых носителях (диски, SSD, ленточные библиотеки) и переживающие перезапуск породившего их процесса или операционной системы.

Постулат (Postulate) — исходное утверждение, принимаемое без строгого формального доказательства в рамках данной теории и служащее базисом для дальнейших логических построений.

Поток (Thread) — наименьшая единица выполнения внутри процесса операционной системы. Потоки одного процесса разделяют общее адресное пространство. Основной поток (Main Thread) в браузере отвечает за рендеринг и исполнение JavaScript.

Превентивные защитные механизмы (Preventive Defense Mechanisms) — меры, направленные на упреждение угрозы до её материализации, а не на реакцию на уже свершившийся инцидент. Пример: лимитирование глубины GraphQL-запроса до его выполнения.

Примат (Primacy) — главенство, первенство одного принципа или подхода над другими в рамках определённой методологии. Примат измеримости означает, что любое оптимизационное решение должно предваряться количественным анализом.

Проактивность (Proactivity) — подход к управлению системой, ориентированный на предвосхищение проблем и их предотвращение, в отличие от реактивного подхода, сфокусированного на устранении последствий уже случившихся инцидентов.

Production-окружение (Production Environment) — среда, в которой программное обеспечение эксплуатируется конечными пользователями для выполнения реальных бизнес-задач. Любые инциденты в этой среде влекут прямые бизнес-последствия.

Р

Растровое изображение (Raster Image) — изображение, представленное в виде двумерной матрицы пикселей, каждому из которых сопоставлено значение цвета. Форматы: JPEG, PNG, WebP, AVIF. В отличие от векторного, при масштабировании теряет качество.

Регрессия (Regression) — непреднамеренное ухудшение существующей функциональности или производительности системы, возникшее в результате внесения изменений в seemingly unrelated часть кодовой базы.

Рекурсивный обход (Recursive Traversal) — алгоритм посещения всех узлов древовидной структуры данных, при котором функция вызывает саму себя для обработки дочерних узлов.

Реляционная база данных (Relational Database, RDBMS) — система управления базами данных, основанная на реляционной модели, где данные представлены в виде таблиц (отношений), а связи между ними — через внешние ключи. Гарантирует ACID-свойства транзакций.

Рендеринг (Rendering) — процесс преобразования структурированного описания интерфейса (DOM + CSSOM) в последовательность пикселей на экране пользовательского устройства, выполняемый браузерным движком.

Репрезентативные данные (Representative Data) — выборка или набор данных, статистически адекватно отражающий свойства и распределение генеральной совокупности, из которой они извлечены. Например, RUM-данные репрезентативны для реальной аудитории.

Релевантные параметры (Relevant Parameters) — подмножество характеристик системы или окружения, оказывающих статистически значимое влияние на изучаемый показатель и потому подлежащих обязательному учёту и контролю.

Репликация (Replication) — процесс создания и поддержания в актуальном состоянии точных копий данных (реплик) на нескольких физически разделённых узлах для повышения доступности и отказоустойчивости.

С

Сегрегация (Segregation) — архитектурный принцип разделения функциональности, ответственности или трафика между различными компонентами системы. Пример: CQRS-сегрегация команд и запросов.

Семантика (Semantics) — смысловое значение, интерпретация формального выражения, метрики или конструкции, в отличие от синтаксиса, описывающего только форму. Семантика метрики LCP — «момент, когда страница визуально полезна».

Сервер (Server) — физический или виртуальный компьютер, предоставляющий вычислительные ресурсы и сервисы (веб, почта, БД) другим компьютерам (клиентам) по сети.

Сетевая конъюнктура (Network Conditions) — текущее состояние сетевой среды, характеризуемое пропускной способностью, задержкой, уровнем потерь пакетов и джиттером.

Синергия (Synergy) — эффект, при котором совокупный результат взаимодействия нескольких компонентов или факторов превышает сумму их индивидуальных эффектов.

Синтетические данные (Synthetic Data) — данные, полученные в результате симуляции или лабораторного эксперимента, а не наблюдения за реальной системой. Используются для тестирования гипотез в контролируемых, воспроизводимых условиях.

Скрипт-маяк (Beacon) — небольшой фрагмент JavaScript-кода, внедряемый в веб-страницу для асинхронной отправки телеметрических данных (метрик RUM) на сервер сбора, как правило, без влияния на пользовательский опыт.

Скролл (Scroll) — вертикальное или горизонтальное перемещение видимой области контента пользователем. События скролла могут использоваться для триггеринга ленивой загрузки или виртуализации.

Сортированное множество (Sorted Set) — структура данных в Redis, хранящая уникальные строки-члены, каждому из которых сопоставлен числовой вес (score), по которому элементы автоматически сортируются.

Список (List) — упорядоченная структура данных, хранящая последовательность элементов. В Redis — тип данных List, реализованный как связный список, оптимизированный для операций вставки и удаления по краям.

Статистически гомогенная выборка (Statistically Homogeneous Sample) — выборка данных, в которой разброс значений вокруг центральной тенденции невелик, а влиянием внешних неконтролируемых факторов можно пренебречь. Характерна для лабораторных, а не полевых измерений.

Стохастика (Stochastics) — раздел математики, изучающий случайные процессы. Стохастическая система — система, поведение которой не полностью детерминировано и может быть описано только в терминах вероятностей.

Т

Таксономия (Taxonomy) — наука о принципах и способах классификации. В контексте пособия — иерархически организованная схема классификации метрик, узких мест или подходов к оптимизации.

Тактовая частота (Clock Rate) — частота синхронизирующих импульсов процессора, измеряемая в ГГц. Определяет, сколько элементарных операций потенциально может быть выполнено за единицу времени, но не является единственным фактором производительности.

Телеметрия (Telemetry) — совокупность технологий и процессов автоматического сбора, передачи и первичной обработки данных о состоянии удалённой системы в реальном времени.

Телеологический процесс (Teleological Process) — процесс, направленный на достижение заранее определённой цели. Телеология оптимизации — достижение целевых значений метрик при минимальных затратах.

Темпоральные метрики (Temporal Metrics) — метрики, измеряющие временные интервалы: длительность загрузки, время отклика, задержку рендеринга. Противопоставляются пространственным метрикам вроде CLS.

Теорема Шеннона-Хартли — фундаментальная теорема теории информации, устанавливающая верхний предел пропускной способности канала связи с аддитивным белым гауссовским шумом: $C = B * \log_2(1 + \frac{S}{N})$

Теория массового обслуживания (Queueing Theory) — раздел прикладной математики, изучающий поведение систем, обрабатывающих поток заявок, с целью прогнозирования длины очередей, времени ожидания и загрузки приборов.

Терминация (Termination) — завершение процесса, соединения или сессии. TLS-терминация — операция расшифровки TLS-трафика на промежуточном узле (балансировщике) для дальнейшей маршрутизации открытого текста внутри доверенного периметра.

Точка присутствия (Point of Presence, PoP) — физическая локация, в которой размещено оборудование CDN-провайдера для обслуживания пользователей из близлежащего географического региона.

Транзакция (Transaction) — последовательность операций с данными, обладающая свойствами ACID: атомарностью, согласованностью, изолированностью и долговечностью.

Трафик (Traffic) — совокупность данных, передаваемых по компьютерной сети за определённый промежуток времени. Измеряется в битах в секунду (бит/с) или пакетах в секунду (pps).

Трейс (Trace) — детализированная запись пути отдельно взятого запроса через все компоненты распределённой системы, включающая временные метки входа и выхода из каждого сервиса. Ключевой артефакт распределённой трассировки.

Троттлинг (Throttling) — механизм принудительного ограничения частоты выполнения операций или потребления ресурсов. Примеры: CPU throttling при перегреве, rate limiting на API-шлюзе.

Тёмные паттерны (Dark Patterns) — элементы пользовательского интерфейса, намеренно спроектированные так, чтобы ввести пользователя в заблуждение и побудить к действиям, которые он не планировал совершать.

У

Узел (Node) — отдельный компьютер, инстанс или процесс, являющийся частью распределённой системы или кластера.

Унификация (Unification) — приведение разнородных форматов, интерфейсов или процессов к единому стандарту для упрощения взаимодействия и обслуживания.

Утилизация ресурсов (Resource Utilization) — степень использования доступного ресурса, обычно выражаемая в процентах от его номинальной мощности (CPU utilization 75%, memory utilization 60%).

Ф

Флагман (Flagship) — наиболее технически совершенный и производительный продукт в линейке производителя (флагманский смартфон, флагманская модель сервера).

Флуктуация (Fluctuation) — случайное отклонение значения физической величины или метрики от среднего уровня. Флуктуации трафика — нормальное явление, требующее запаса по пропускной способности.

Фреймворк (Framework) — программная платформа, определяющая структуру и предоставляющая набор библиотек и инструментов для разработки

приложений определённого класса, накладывающая архитектурные соглашения (React, Vue, Django, Express.js).

Фрустрация (Frustration) — негативное эмоциональное состояние пользователя, возникающее вследствие расхождения между ожидаемой и фактической отзывчивостью интерфейса. Является прямым предиктором отказа от сервиса.

Х

Хук (Hook) — специальная функция в современных фронтенд-фреймворках (React, Vue), позволяющая подключаться к внутренним механизмам (состояние, жизненный цикл, побочные эффекты) из функциональных компонентов без использования классового синтаксиса.

Хэндшейк (Handshake) — процедура предварительного обмена служебными сообщениями между двумя устройствами для согласования параметров соединения. TCP-хэндшейк — трёхэтапный; TLS-хэндшейк — согласование шифров и сертификатов.

Ц

Центр сертификации (Certificate Authority, CA) — доверенная организация, выпускающая и подписывающая цифровые сертификаты открытых ключей, используемые для аутентификации сторон при установлении TLS-соединения.

Централизация (Centralization) — архитектурный принцип, при котором управление, принятие решений или хранение данных сосредоточено в едином компоненте системы.

Ч

Чанк (Chunk) — отдельный файл, полученный в результате разделения (code splitting) исходного бандла. Загружается браузером асинхронно по мере необходимости.

Ш

Шардирование (Sharding) — техника горизонтального партиционирования данных, при которой строки одной логической таблицы распределяются по нескольким физическим серверам (шардам) на основе значения ключа шардирования.

Шина (Bus) — подсистема передачи данных между функциональными блоками компьютера (процессором, памятью, периферией). Характеризуется разрядностью и пропускной способностью.

Э

Эвристический инструментарий (Heuristic Toolkit) — набор методов и приёмов, основанных на эвристиках — правилах, полученных из практического опыта и дающих приемлемое решение в большинстве случаев, но не гарантирующих оптимальности или абсолютной точности.

Эксплицитное определение (Explicit Definition) — строгое, формально изложенное определение понятия, не допускающее двусмысленного толкования и не опирающееся на неявные контекстные допущения.

Элиминация шума (Noise Elimination) — процесс очистки данных от случайных, неинформативных флуктуаций с целью выделения значимого сигнала (тренда, аномалии). Достигается методами фильтрации, агрегации и статистической обработки.

Эндпоинт (Endpoint) — конкретный URL или URI, по которому API принимает запросы. Каждый эндпоинт ассоциирован с определённой операцией или ресурсом.

Эталон (Baseline, Reference) — зафиксированный, верифицированный набор показателей или конфигураций, служащий точкой отсчёта для сравнения при оценке эффективности изменений.

A

Apache Kafka — распределённая, горизонтально масштабируемая платформа потоковой передачи событий, оптимизированная для высокопроизводительного обмена сообщениями между продюсерами и консюмерами с гарантией сохранности.

API (Application Programming Interface) — программный интерфейс, набор правил и спецификаций, посредством которого один программный компонент может взаимодействовать с другим.

B

Bandwidth-Delay Product (BDP) — произведение пропускной способности канала на задержку кругового обхода (RTT). Определяет объём данных, который должен находиться «в полёте» для полной утилизации канала.

Brotli — алгоритм сжатия данных общего назначения, разработанный Google, демонстрирующий на 20–26% более высокую степень компрессии текстовых ресурсов по сравнению с Gzip при сопоставимых вычислительных затратах на распаковку.

C

CDN (Content Delivery Network) — географически распределённая сеть серверов, предназначенная для быстрой доставки статического и, в современных реализациях, динамического контента конечным пользователям за счёт обслуживания запросов с ближайшей точки присутствия.

CI/CD (Continuous Integration / Continuous Delivery) — практика автоматизации сборки, тестирования и развёртывания программного обеспечения, обеспечивающая частую и надёжную доставку изменений в продуктивное окружение.

Code Splitting — техника разделения кодовой базы на отдельные фрагменты (чанки), загружаемые по требованию, для сокращения времени начальной загрузки приложения.

Core Web Vitals — инициатива Google, определяющая три ключевые метрики пользовательского опыта: LCP (скорость загрузки основного контента), INP (отзывчивость взаимодействий) и CLS (визуальная стабильность).

Critical Rendering Path (CRP) — последовательность шагов, выполняемых браузером для первичной отрисовки контента на экране: построение DOM и CSSOM, формирование Render Tree, компоновка (Layout) и отрисовка (Paint).

CRUD (Create, Read, Update, Delete) — акроним, обозначающий четыре базовые операции с данными: создание, чтение, обновление и удаление. Лежит в основе большинства REST API и интерфейсов взаимодействия с БД.

CSSOM (CSS Object Model) — объектная модель, представляющая собой древовидную структуру всех CSS-правил, построенную браузером в процессе парсинга каскадных таблиц стилей. Является блокирующим рендеринг компонентом.

D

DOM (Document Object Model) — объектная модель документа, представляющая HTML-документ в виде древовидной структуры узлов, доступной для программного чтения и модификации через JavaScript.

DoS (Denial of Service) — атака, направленная на отказ в обслуживании, при которой легитимные пользователи теряют доступ к ресурсу вследствие исчерпания его вычислительных или сетевых ресурсов.

DDoS (Distributed Denial of Service) — распределённая атака типа «отказ в обслуживании», осуществляемая одновременно с множества географически разнесённых устройств (ботнетов), что делает её значительно сложнее для фильтрации.

E

Event Loop (Цикл событий) — архитектурный паттерн в JavaScript и Node.js, обеспечивающий неблокирующую асинхронную обработку операций в одном потоке путём циклического опроса очередей микро- и макрозадач.

G

Git-система (Git Version Control System) — распределённая система контроля версий, позволяющая отслеживать изменения в файлах, координировать работу нескольких разработчиков и обеспечивать воспроизводимость сборок.

gRPC — высокопроизводительный RPC-фреймворк от Google, использующий Protocol Buffers (protobuf) для бинарной сериализации данных и HTTP/2 в качестве транспортного протокола. Оптимизирован для межсервисного взаимодействия.

GraphQL — язык запросов к API, позволяющий клиенту специфицировать точную структуру требуемых данных в одном запросе, что решает проблемы избыточности (over-fetching) и недостаточности (under-fetching) выборок.

Gzip — широко распространённый алгоритм сжатия данных без потерь, обеспечивающий хороший баланс между степенью компрессии и скоростью работы. Повсеместно поддерживается веб-серверами и браузерами.

H

HTTP (HyperText Transfer Protocol) — протокол прикладного уровня, лежащий в основе передачи данных в World Wide Web.

HTTPS (HTTP Secure) — расширение протокола HTTP, использующее TLS для шифрования и аутентификации передаваемых данных.

J

JSON (JavaScript Object Notation) — легковесный текстовый формат сериализации структурированных данных, легко читаемый человеком и машиной. Является стандартным форматом обмена данными в REST API.

L

Lighthouse — инструмент от Google для аудита веб-страниц, измеряющий производительность, доступность, SEO и Best Practices. Генерирует лабораторные отчёты в контролируемой среде.

M

MobX — библиотека управления состоянием, основанная на принципах реактивного программирования и автоматическом отслеживании зависимостей между наблюдаемыми данными и вычисляемыми производными.

P

Preview-окружение (Preview Environment) — временное, изолированное окружение, автоматически создаваемое для каждого pull request'a, позволяющее провести ручное тестирование и синтетические аудиты до слияния изменений в основную ветку.

R

RabbitMQ — брокер сообщений с открытым исходным кодом, реализующий протокол AMQP. Используется для асинхронной связи между сервисами через очереди сообщений.

Redux — библиотека управления состоянием для JavaScript-приложений, основанная на принципах Flux и использующая единое иммутабельное хранилище (store) и чистые функции-редюсеры.

REST API (Representational State Transfer API) — архитектурный стиль проектирования API, оперирующий ресурсами с уникальными URI и использующий стандартные HTTP-методы (GET, POST, PUT, DELETE) для операций над ними.

Round-Trip Time (RTT) — время, необходимое пакету данных для прохождения пути от отправителя к получателю и обратно. Фундаментальная характеристика сетевого канала, определяющая минимальную достижимую латентность.

S

Server-Sent Events (SSE) — технология односторонней потоковой передачи данных от сервера к клиенту по инициативе сервера через стандартное HTTP-соединение. В отличие от WebSockets, не поддерживает двусторонний обмен.

Service Level Objective (SLO) — конкретное числовое значение целевого уровня надёжности или производительности сервиса, которое команда обязуется поддерживать (например, P95 Latency < 200 мс).

SQL (Structured Query Language) — декларативный язык программирования, предназначенный для управления и манипулирования данными в реляционных СУБД.

T

TCP (Transmission Control Protocol) — протокол транспортного уровня, обеспечивающий гарантированную, упорядоченную доставку потока байтов между узлами с контролем ошибок и перегрузок.

Terser — JavaScript-парсер, минификатор и компрессор, используемый по умолчанию в Webpack 5 для production-сборок. Выполняет удаление неиспользуемого кода, обфускацию и сжатие.

TLS (Transport Layer Security) — криптографический протокол, обеспечивающий защищённую передачу данных между узлами в сети путём шифрования, аутентификации и контроля целостности.

Tree Shaking — процесс удаления неиспользуемого кода (dead code) из финального бандла на этапе сборки, основанный на статическом анализе графа импортов ES-модулей.

U

UDP (User Datagram Protocol) — протокол транспортного уровня, передающий датаграммы без установления соединения, гарантии доставки и контроля порядка. Обеспечивает минимальную латентность ценой надёжности.

UI (User Interface) — пользовательский интерфейс, совокупность визуальных элементов и средств взаимодействия, через которые пользователь управляет программой.

UX (User Experience) — пользовательский опыт, совокупность субъективных впечатлений и ощущений, возникающих у пользователя при взаимодействии с продуктом.

V

Viewport — видимая область веб-страницы в окне браузера без прокрутки. Размеры viewport являются ключевым фактором при адаптивной вёрстке и расчёте метрики CLS.

Vite — современный инструмент сборки фронтенд-проектов, использующий нативные ES-модули для мгновенного запуска dev-сервера и Rollup для production-сборки. Отличается высокой скоростью работы.

W

Webpack — наиболее распространённый модульный сборщик для JavaScript-приложений, преобразующий и объединяющий модули и зависимости в оптимизированные статические артефакты (бандлы).

WebSockets — протокол полнодуплексной связи поверх TCP-соединения, позволяющий серверу и клиенту обмениваться сообщениями в реальном времени без повторных HTTP-запросов.

Z

Zustand — легковесная библиотека управления состоянием для React, основанная на хуках и иммутабельных обновлениях, не требующая провайдеров и сложной конфигурации.

Zero Round-Trip Time (0-RTT) — режим возобновления соединения в протоколах TLS 1.3 и QUIC, позволяющий начать передачу прикладных данных в первом же пакете от клиента, без предварительных циклов рукопожатия.

СОДЕРЖАНИЕ

Учебное издание

Илья Сергеевич Поломошнов,
Оксана Андреевна Литвиненко

Оптимизация веб-приложений

Учебное пособие

Редактор О.А.Литвиненко
Компьютерная верстка И.С.Поломошнова

Свидетельство о государственной регистрации
№2670 от 11 августа 2017 г.

Подписано в печать «__»_____ 2026 г. Формат _____
Уч.-изд. л. ____ Усл. печ. л. ____ Тираж ___ экз. Зака ___
Цена свободная.

Издательство Бурятского госуниверситета имени Доржи Банзарова
670000, г. Улан-Удэ, ул. Смолина, 24а
rio@bsu.ru

Отпечатано в типографии Издательства
Бурятского госуниверситета имени Доржи Банзарова
67000, г. Улан-Удэ, ул. Сухэ-Батора, 3а