System Design

пережить интервью

Чжиюн Тань

Предисловие
Энтони Асты и Майкла Элдера

MANNING

Acing the System Design Interview

ZHIYONG TAN

MANNING
SHELTER ISLAND

System Design пережить интервью

ЧЖИЮН ТАНЬ

Выпущено
при поддержке
КРОК

ПИТЕР®
Санкт-Петербург • Москва • Минск
2025

ББК 32.973.2-018
УДК 004.3
Ч-57

Чжиюн Тань

Ч-57 System Design: пережить интервью. — СПб.: Питер, 2025. — 544 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-4106-7

Собеседования по проектированию систем могут стать серьезной преградой даже для опытных разработчиков. К счастью, стоит немного подготовиться — и все эти вопросы, не имеющие однозначного ответа, и whiteboard-сессии станут вашим конкурентным преимуществом! В своей замечательной книге Чжиюн Тань делится практикой успешных собеседований и советами по дизайну систем, благодаря которым разработчики получали предложения от Amazon, Apple, ByteDance, PayPal и Uber.

«System design: пережить интервью» — мастер-класс по уверенному прохождению собеседования. Используя простые и легко запоминающиеся методы, вы научитесь быстро анализировать задачи, находить эффективные решения и четко объяснять свои идеи эксперту. В ходе работы с книгой вы не только приобретете навыки, необходимые, чтобы успешно пройти собеседование, но и попрактикуетесь в создании качественного дизайна.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018
УДК 004.3

ISBN 978-1633439108 англ.

978-5-4461-4106-7

Краткое содержание

Часть 1

Часть 2

Оглавление

Часть 2

Посвящаю своим маме и папе

Предисловие

Последние 20 лет я занимался формированием команд инженеров распределенных систем в крупнейших технологических компаниях (Google, Twitter и Uber). По моему опыту, эффективность команд в этих компаниях обусловлена умением выявлять на собеседованиях талантливых инженеров, обладающих мастерством проектирования систем, которое они демонстрируют экспертам. «System Design: пережить интервью» — бесценное практическое руководство, которое поможет как перспективным разработчикам, так и опытным профессионалам приобрести знания и навыки, необходимые, чтобы показать отличные результаты на техническом собеседовании. В условиях, когда на первый план выходит умение проектировать масштабируемые и надежные системы, эта книга станет настоящей сокровищницей ценных наблюдений, стратегий и практических рекомендаций, которые несомненно помогут читателю разобраться в тонкостях system design interview, посвященного проектированию систем.

Спрос на стабильные и масштабируемые системы продолжает расти, и при подборе сотрудников компании все больше внимания уделяют опыту проектирования систем. Эффективное собеседование оценивает не только техническую квалификацию кандидата, но и его способность критически мыслить, принимать обоснованные решения и решать сложные задачи.

Чжиюн — опытный разработчик, и благодаря своему ви́дению и глубокому пониманию принципов собеседования, посвященного проектированию систем, он станет идеальным проводником для каждого, кто захочет освоить этот важнейший набор навыков.

В этой книге Чжиюн представляет подробный план, который проведет читателя по всем этапам собеседования. Начав с обзора основных принципов и концепций, он переходит к подробному рассмотрению проблем проектирования, включая масштабирование, надежность, производительность и управление данными. Он разбивает каждую тему на простые и понятные составляющие,

давая четкие пояснения и сопровождая их реальными практическими примерами. Имеющийся опыт прохождения интервью с отраслевыми экспертами позволяет ему сорвать покров таинственности с процесса собеседования. Он делится ценными наблюдениями о психологии экспертов, а также о часто встречающихся вопросах и ключевых факторах, учитываемых экспертами при оценке кандидата. Так он не только помогает читателям понять, чего ожидать во время собеседования, но и вселяет в них уверенность и наделяет инструментами, необходимыми, чтобы добиться успеха в условиях, когда ставки очень высоки.

Объединение теории, содержащейся в части 1, с практической частью 2 позволяет читателям не только усвоить теоретические основы, но и выработать умение применять эти знания в реальных условиях. Кроме того, книга не ограничивается техническими вопросами и подчеркивает важность эффективных коммуникаций в процессе собеседования по проектированию систем. Чжиюн исследует, как эффективно выражать свои мысли, представлять решения и участвовать в совместной работе с экспертами. Этот комплексный подход основан на том факте, что успешное проектирование систем зависит не только от технической одаренности, но и от умения доносить свои мысли и взаимодействовать с окружающими.

Независимо от того, готовитесь ли вы к собеседованию или хотите повысить свою квалификацию в области проектирования систем, эта книга станет вашим верным спутником, который поможет уверенно и элегантно справляться даже с самыми сложными проблемами проектирования.

Итак, погружайтесь в материал, получайте новые знания и навыки и отправляйтесь в путь к заветной цели — искусству создания масштабируемых и надежных систем. Вы несомненно станете ценным сотрудником в любой организации и сделаете успешную карьеру разработчика ПО.

Собеседование по проектированию систем ждет вас!

Энтони Аста (Anthony Asta),
технический директор LinkedIn
(ранее занимавший технические руководящие должности
в Google, Twitter и Uber)

В мире разработки ПО непрерывно *всё*. Непрерывные улучшения, непрерывная доставка, непрерывный мониторинг и непрерывная переоценка потребностей пользователя, а также вычислительной мощности и производительности — неотъемлемые признаки любой серьезной программной системы. Если вы хотите стать успешным разработчиком, вы должны постоянно обучаться и расти. Энтузиасты нашей отрасли определяют даже то, как участники сообщества

взаимодействуют друг с другом, как обмениваются знаниями и как формируют свой стиль жизни.

Тренды разработки постоянно меняются, от популярных языков программирования и фреймворков до программируемых облачных инфраструктур. Если вы, как и я, работаете в этой отрасли уже несколько десятков лет, вы можете наблюдать, как эти тренды сменяют друг друга. Лишь одно остается неизменным: умение логически и аргументированно рассуждать о том, как программная система работает, организует свои данные и взаимодействует с людьми, абсолютно необходимо, чтобы стать эффективным разработчиком или техническим руководителем.

Будучи инженером-разработчиком (и дойдя до позиции Distinguished Engineer в IBM), я своими глазами видел, как от компромиссов проектирования зависит успех или неудача программной системы. Кем бы вы ни были — новичком, желающим получить свою первую должность, или закаленным ветераном, ищущим новых интересных задач в новой компании, — эта книга поможет вам глубже разобраться в логических связях, лежащих в основе компромиссов, и научиться их объяснять.

Книга обобщает и упорядочивает многие вопросы, которые необходимо учитывать при проектировании любых программных систем. Чжиюн Тань подготовил блестящий краткий курс по основным компромиссам проектирования систем и представил множество практических примеров, которые помогут подготовиться даже к самым сложным собеседованиям.

Часть 1 начинается с содержательного обзора важнейших вопросов проектирования систем. Начиная с нефункциональных требований, вы узнаете о типичных факторах, которые необходимо учитывать при обдумывании компромиссов. Далее вы подробно рассмотрите составление спецификации API, поясняющей, как дизайн вашей системы на практике решает задачу, поставленную на собеседовании. Разобравшись с API, вы изучите некоторые лучшие практики организации модели данных системы с применением хранилищ данных, которые считаются отраслевым стандартом, и паттернов управления распределенными транзакциями. Помимо рассмотрения самых распространенных практических сценариев, вы узнаете о ключевых аспектах работы системы, включая современные подходы к наблюдаемости и управлению ведением журналов.

В части 2 рассматриваются 11 разных задач проектирования систем, от обмена текстовыми сообщениями до Airbnb. Разбирая каждую из них, вы узнаете что-то новое о том, как формулировать правильные вопросы для упорядочения нефункциональных системных требований и какие компромиссы стоит анализировать для ее решения. Проектирование систем — набор практических навыков, в основе которых лежит опыт, а его, в свою очередь, можно получить, обучаясь на примерах других. Если вы усвоите уроки и премудрости, заложенные в примерах

из книги, на собеседованиях по проектированию систем вы будете готовы даже к самым сложным задачам.

Вклад, который Чжиюн Тань внес в нашу отрасль этой работой, произвел на меня глубокое впечатление. Беретесь ли вы за эту книгу, только получив диплом или уже проработав в отрасли много лет, я надеюсь, что вы найдете в ней возможности для личного роста — как это случилось со мной.

Майкл Д. Элдер (Michael D. Elder)
Distinguished Engineer и Senior Director, компания PayPal
(ранее IBM Distinguished Engineer и IBM Master Inventor,
компания IBM)

Введение

Среда, около 16 часов. У вас только что закончилось последнее видеособеседование в компании мечты, и вас переполняют знакомые ощущения: усталость, разочарование и чувство дежавю. Вы уже знаете, что через пару дней получите сообщение, которое видели за свою многолетнюю карьеру не один раз. «Спасибо за ваш интерес к позиции старшего разработчика в XXX. Мы тщательно рассмотрели вашу кандидатуру, и несмотря на ваш впечатляющий опыт и практические навыки, к сожалению, мы не можем предложить вам эту должность».

Это снова было собеседование по проектированию систем. Вам поручили создать приложение фотохостинга, и вы предложили действительно замечательный дизайн — масштабируемый, гибкий и простой в обслуживании. В нем использовались новейшие фреймворки и лучшие практики жизненного цикла разработки. Но вы видели, что эксперта это не впечатлило. Отстраненное выражение его лица и утомленный, спокойный, вежливый тон ясно говорили, что он вел себя как положено профессионалу, чтобы «у кандидата остался позитивный опыт».

Это ваша седьмая попытка устроиться в эту компанию за четыре года; вы также не раз проходили собеседования в других компаниях, в которых вам действительно хотелось бы работать. Вы мечтаете стать частью этой компании с миллиардами пользователей; она создает ведущие в отрасли фреймворки и языки программирования. Вы знаете, что люди, которых вы встретите, и знания, которые получите в этой компании, принесут огромную пользу вашей карьере, и эта работа будет лучшим вложением вашего времени.

При этом вы неоднократно получали повышение в других компаниях, где работали ранее. Сейчас вы сеньор-разработчик, поэтому вам еще обиднее, что вы не можете пройти собеседование на такую же должность в компании своей мечты. Вы были техническим руководителем многих проектов, возглавляли и обучали группы джунов, строили и обсуждали системные архитектуры со стафф- и сеньор-инженерами, вносили заметный и полезный вклад в проектирование многих систем. Перед каждым собеседованием в компании своей мечты вы читали кучу постов в инженерных блогах и смотрели все видеолекции,

вышедшие за последние три года. Вы также прочли множество авторитетных изданий о микросервисах, приложениях для обработки больших объемов данных, паттернах облачных платформ и предметно-ориентированном проектировании (domain-driven design, DDD). Почему же вы никак не можете пройти эти собеседования?

Может, вам просто не везло? Соотношение «спрос/предложение» кандидатов не соответствовало количеству вакансий в этих компаниях? Подвела вероятность того, что выберут именно вас? Вы участвуете в лотерее? Нужно просто повторять попытки каждые полгода, пока вам не повезет? А может, воскурить благовония и принести более щедрые подношения богам собеседований/ревью?

Вы делаете глубокий вдох и закрываете глаза, чтобы спокойно подумать. Становится понятно, что за 45 минут, выделенные вам для обсуждения дизайна системы, можно очень многое улучшить. (Каждое собеседование занимает час, но если убрать знакомство и вопросы/ответы, у вас фактически остается 45 минут на дизайн сложной системы, которая обычно эволюционирует с течением времени.)

Беседа с коллегами-инженерами подтверждает ваше предположение. Вы недостаточно четко выяснили требования к системе. Вы считали, что от вас требуют построить минимально жизнеспособный продукт для бэкенда, который обслуживает задачи хранения и организации совместного доступа к фотографиям в мобильном приложении, и начали набрасывать примерную спецификацию API. Эксперту пришлось прервать вас и указать, что система должна масштабироваться для миллиарда пользователей. Вы нарисовали схему системной архитектуры, которая включала CDN, но не обсудили плюсы и минусы своих решений, а также альтернативные варианты. Вы не предложили возможности за пределами области, которую эксперт обрисовал в начале собеседования, — например, аналитический механизм для определения самых популярных фото или персонализированные рекомендации фотографий. Вы не задали правильные вопросы и не упомянули такие важные концепции, как ведение журналов, мониторинг и оповещения.

Вы осознаете, что даже со всем вашим опытом разработки, постоянным обучением и стремлением быть в курсе лучших практик и достижений отрасли проектирование систем невероятно обширно и вам не хватает формальных знаний и понимания многих компонентов, с которыми вы не соприкасаетесь напрямую (например, балансировщиков нагрузки или некоторых баз данных NoSQL). А значит, вы не сможете создать схему системной архитектуры того уровня полноты, которого ожидает эксперт, и оперативно масштабировать обсуждение разных уровней системы. Пока вы не научитесь это делать, вы не достигнете приемлемого порога предложений о работе; вы не сможете в полной мере понять сложную систему или занять более высокую техническую или руководящую должность.

Благодарности

Благодарю свою жену Эмму за то, что она постоянно поддерживала меня в разных начинаниях, погружении в трудные и времязатратные проекты, написании приложений и подготовке этой книги. Спасибо моей дочери Аде — она вдохновляла меня и помогала справиться с раздражением и усталостью от программирования и писательства.

Спасибо моему брату Чжилуну за такую полезную обратную связь по моим черновикам; он сам является экспертом в проектировании систем и протоколах кодирования видео в Meta. Спасибо моей старшей сестре Сюминь, которая всегда поддерживала меня и подталкивала к новым достижениям.

Спасибо маме и папе за все, чем они пожертвовали, чтобы эта работа стала возможной.

Хочу поблагодарить за помощь сотрудников издательства Manning. Благодарю рецензентов моей книги Андреаса фон Линдена (Andreas von Linden), Амутана Ганешана (Amuthan Ganeshan), Марка Рулло (Marc Roulleau), Дина Цалтаса (Dean Tsaltas) и Винсента Лайарда (Vincent Liard). Амутан представил подробный отзыв и задал правильные вопросы по предложенным темам. Кэти Спозато Джонсон (Katie Sposato Johnson) была моим наставником на протяжении полутора лет рецензирования и переработки рукописи. Она внимательно прочитала каждую главу, и ее замечания значительно улучшили стиль и четкость изложения. Мой научный редактор, Мохит Чилкоти (Mohit Chilkoti), внес много дельных предложений для улучшения ясности и указал на ошибки. Мой редактор Адриана Сабо (Adriana Sabo) и ее команда организовали панельные обсуждения, собравшие бесценную обратную связь, благодаря которой эта книга стала значительно лучше. Спасибо всем рецензентам: Абдулу Кариму Мемону (Abdul Karim Memon), Аджиту Маллери (Ajit Malleri), Алессандро Буггину (Alessandro Buggin), Алессандро Кампейсу (Alessandro Campeis), Андресу Сакко (Andres Sacco), Анто Аравинту (Anto Aravinth), Ашвини Гупте (Ashwini Gupta), Клиффорду Турберу (Clifford Thurber), Кертису Вашингтону (Curtis Washington), Дипкумару Пателу (Dipkumar Patel), Фасиху Хатибу (Fasih Khatib),

Ганешу Сваминатану (Ganesh Swaminathan), Хаиму Раману (Haim Raman), Харешу Лале (Haresh Lala), Джавиду Асгарову (Javid Asgarov), Йенсу Кристиану Б. Мэдсену (Jens Christian B. Madsen), Джереми Чену (Jeremy Chen), Джону Риддлу (Jon Riddle), Джонатану Ривзу (Jonathan Reeves), Камешу Ганесану (Kamesh Ganesan), Кирану Ананте (Kiran Anantha), Лауду Бентилу (Laud Bentil), Лоре Вардаровой (Lora Vardarova), Мэтту Фердереру (Matt Ferderer), Максу Садрие (Max Sadrieh), Майку Б. (Mike B.), Мунибу Шайху (Muneeb Shaikh), Наджибу Арифу (Najeeb Arif), Нарендрану Солай Шридхарану (Narendran Solai Sridharan), Нолану То (Nolan To), Нурану Махмуду (Nouran Mahmoud), Патрику Ванджау (Patrick Wanjau), Пейти Ли (Peiti Li), Петеру Шабо (Péter Szabó), Пьеру-Мишелю Анселу (Pierre-Michel Ansel), Прадипу Челлаппану (Pradeep Chellappan), Рахулу Модпуру (Rahul Modpur), Раджешу Моханану (Rajesh Mohanan), Садхане Ганапатираджу (Sadhana Ganapathiraju), Самсону Хаилу (Samson Hailu), Сэмюэлю Бошу (Samuel Bosch), Сандживу Килапару (Sanjeev Kilarapu), Симеону Лейзерзону (Simeon Leyzerzon), Шраванти Редди (Sravanthi Reddy), Винсенту Нго (Vincent Ngo), Зохебу Айнапору (Zoheb Ainapore), Зородзаю Мукуе (Zorodzayi Mukuya) — ваши предложения помогли сделать эту книгу лучше.

Хочу поблагодарить Марка Рулло, Андреса фон Линдена, Амутана Ганешана, Роба Конери (Rob Conery) и Скотта Хансельмана (Scott Hanselman) за их поддержку и рекомендации дополнительных источников информации.

Спасибо суровым северянам Эндрю Уолдрону (Andrew Waldron) и Иэну Хоу (Ian Hough). Энди убедил меня добавить ряд полезных технических деталей во все главы, а также помог разобраться, как правильно форматировать рисунки под размер страниц. Он помог мне понять, что я способен сделать гораздо больше, чем думал. Айра Душич (Aira Dučić) и Матко Хрватин (Matko Hrvatin) очень помогли с маркетингом, а Драгана Бутиган-Берберович (Dragana Butigan-Berberović) и Айвен Мартинович (Ivan Martinović) отлично справились с форматированием. Степан Джурекович (Stjepan Jureković) и Никола Димитриевич (Nikola Dimitrijević) помогали с промороликом к книге.

О книге

Эта книга посвящена веб-сервисам. Кандидат должен обсудить требования к системе, а затем спроектировать ее с соответствующим уровнем сложности и затрат.

Кроме собеседований по написанию кода, на большинство позиций в области программной инженерии и архитектур и на руководящие инженерные должности проводятся собеседования по проектированию систем.

Умение проектировать и анализировать крупномасштабные системы становится все более важным с ростом уровня позиции. Соответственно, собеседования по проектированию систем получают больший вес при приеме на руководящие должности. Подготовка к собеседованиям (как в качестве эксперта, так и в качестве кандидата) станет хорошим вложением времени для карьеры в технологической сфере.

Открытая природа собеседований по проектированию систем существенно усложняет подготовку к ним, поскольку невозможно заранее узнать, как и что будет обсуждаться во время собеседования. Более того, найти книги, посвященные этой теме, трудно. Дело в том, что проектирование систем — искусство и наука одновременно. Его суть не в достижении совершенства, а в принятии компромиссного варианта дизайна, который можно реализовать с имеющимися ресурсами и временем и который будет наиболее близок к текущим и возможным будущим требованиям. Эта книга поможет читателю заложить фундамент будущих знаний или выявить и заполнить пробелы в имеющихся.

Собеседование по проектированию систем также проверяет навыки вербального общения, сообразительности, умения задавать правильные вопросы и справляться со страхом возможной неудачи. Эта книга подчеркивает важность умения эффективно и лаконично описать свой опыт проектирования систем в ходе собеседования, занимающего менее одного часа, и направить собеседование в нужное русло, задавая эксперту правильные вопросы. Чтение этой книги наряду с отработкой навыков проектирования систем совместно

с другими инженерами позволит вам получить необходимые знания и свободно овладеть материалом, чтобы пройти собеседование и плодотворно участвовать в проектировании систем в организации, к которой вы присоединяетесь. Книга также станет полезным ресурсом для экспертов, проводящих собеседования по проектированию систем.

ДЛЯ КОГО ЭТА КНИГА

Книга предназначена для разработчиков, специалистов по программным архитектурам и технических руководителей, желающих вывести свою карьеру на новый уровень.

Это не вводный курс проектирования и разработки. Лучше читать эту книгу, уже имея минимальный опыт работы в отрасли: например, студенту-стажеру может понадобиться изучить документацию и другие материалы о тех инструментах, которые ему незнакомы, и обсудить их наряду с другими непонятными концепциями из этой книги с коллегами-инженерами. В книге обсуждается подход к собеседованиям по проектированию систем; она не повторяет вводный материал, который можно найти в интернете или в других книгах. От читателя ожидаются определенные навыки программирования и владения SQL.

СТРУКТУРА КНИГИ

Книга состоит из двух частей, разделенных на 17 глав, и четырех коротких приложений.

Часть 1 напоминает учебник. В главах, составляющих эту часть, рассматриваются темы, поднимаемые на собеседованиях по проектированию систем.

Часть 2 состоит из обсуждений типичных вопросов, относящихся к концепциям из части 1, которые задают на собеседованиях. В каждой главе используются концепции, рассмотренные в части 1. Книга ориентируется на общие веб-сервисы; в нее не включены специализированные и сложные темы: платежи, видеостримы, сервисы геолокации или разработка баз данных. Более того, по моему мнению, когда эксперт просит кандидата 10 минут обсуждать линеаризуемость базы данных или вопросы согласованности (такие, как сервисы координации, кворум или gossip-протоколы), он узнает о его опыте лишь то, что тот прочитал достаточно, чтобы обсуждать указанную тему в течение 10 минут. Для собеседований на должности, требующие опыта в специфических областях, должны быть отдельные книги. А в этой книге при упоминании таких тем будут указываться другие книги или ресурсы, им посвященные.

ФОРУМ LIVEBOOK

Приобретая книгу «System Design: пережить интервью», вы получаете бесплатный доступ к закрытому веб-форуму издательства Manning (на английском языке), на котором можно оставлять комментарии о книге, задавать технические вопросы и получать помощь от автора и других пользователей. Чтобы получить доступ к форуму, откройте страницу https://livebook.manning.com/book/acing-the-system-design-interview/discussion. Информацию о форумах Manning и правилах поведения на них см. на https://livebook.manning.com/discussion.

В рамках своих обязательств перед читателями издательство Manning предоставляет ресурс для содержательного общения читателей и авторов. Эти обязательства не подразумевают конкретную степень участия автора, которое остается добровольным (и неоплачиваемым). Задавайте автору хорошие вопросы, чтобы он не терял интереса к происходящему! Форум и архивы обсуждений доступны на веб-сайте издательства, пока книга продолжает издаваться.

Другие ресурсы в интернете:

- https://github.com/donnemartin/system-design-primer
- https://bigmachine.io/products/mission-interview/
- http://geeksforgeeks.com
- http://algoexpert.io
- https://www.learnbay.io/
- http://leetcode.com
- https://bigmachine.io/products/mission-interview/

Об авторе

Чжиюн Тань занимает позицию менеджера в PayPal. Ранее он работал сеньор-full-stack-инженером в Uber, инженером-разработчиком в Teradata и инженером по работе с данными в стартапах. Ему доводилось сидеть по разные стороны стола на многочисленных собеседованиях по проектированию систем. Чжиюн также получал предложения о работе от таких перспективных компаний, как Amazon, Apple и ByteDance/TikTok.

О научном редакторе

Мохит Чилкоти — специалист по платформенным архитектурам в Chargebee. Он является обладателем сертификата AWS Solutions Architect; в частности, он проектировал торговую платформу Alternative Investment Trading Platform для Morgan Stanley и платформу розничных продаж для Tekion Corp.

Иллюстрация на обложке

Иллюстрация под названием Femme Tatarsk Tobolsk («Татарка из Тобольска»), помещенная на обложку книги, взята из вышедшего в 1784 году каталога национальных костюмов, составленного Жаком Грассе де Сен-Совером (Jacques Grasset de Saint-Sauveur). Каждая иллюстрация этого каталога тщательно прорисована и раскрашена от руки.

В прежние времена по одежде человека было легко определить, где он живет и каковы его профессия или положение в обществе. Manning отдает дань изобретательности и инициативности современной компьютерной отрасли, используя для своих изданий обложки, которые демонстрируют богатое вековое разнообразие региональных культур, оживающее на изображениях из собраний, подобных этому.

От издательства

Мы выражаем огромную благодарность компании КРОК за помощь в работе над русскоязычным изданием книги и их вклад в повышение качества переводной литературы.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

О НАУЧНОМ РЕДАКТОРЕ РУССКОГО ИЗДАНИЯ

Александр Петраки — старший инженер-разработчик компании КРОК. Занимается проектированием архитектуры высоконагруженных приложений и выполняет реализацию back-end-части на Java и Spring с применением СУБД MySQL, PostgreSQL, Oracle, JanusGraph.

Часть 1

В этой части рассматриваются общие темы, обсуждаемые на собеседованиях по проектированию систем. Она готовит читателя к части 2, в которой представлены примеры типичных задаваемых вопросов.

В главе 1 мы начнем с краткого разбора примера системы и попутно представим концепции проектирования систем, подробно в них не вдаваясь. Все эти концепции будут детально рассматриваться в следующих главах.

В главе 2 обсудим личный опыт типичных собеседований по проектированию систем. Вы научитесь прояснять требования задачи и определять, какие части системы можно оптимизировать за счет остальных. Затем рассмотрим другие популярные темы, включая хранение и поиск данных, вспомогательные операции, такие как мониторинг и оповещения, а также граничные случаи и новые ограничения.

В главе 3 речь пойдет о нефункциональных требованиях, которые обычно не формулируются явно заказчиком или экспертом на собеседовании; их необходимо прояснить перед началом проектирования системы.

Большая система может обслуживать сотни миллионов пользователей, ежедневно получая миллиарды запросов на чтение и запись данных. В главе 4 вы узнаете, как масштабировать базы данных, чтобы они справлялись с таким трафиком.

Система может быть разделена на сервисы; возможно, вам придется записывать связанные данные в разные сервисы. Эта тема обсуждается в главе 5.

Многие системы требуют наличия общей функциональности. В главе 6 речь пойдет о том, как централизовать такую сквозную функциональность в сервисы, способные обслуживать многие другие системы.

1 Обзор основных концепций проектирования систем

В ЭТОЙ ГЛАВЕ

- ✓ Важность собеседований по проектированию систем
- ✓ Масштабирование сервисов
- ✓ Облачный хостинг или размещение на физических серверах

Собеседование по проектированию систем проходит в форме диалога между соискателем и экспертом, посвященного проектированию программной системы, обычно предоставляющей свою функциональность по сети. В начале интервью эксперт кратко и в общих чертах формулирует для кандидата задачу по проектированию конкретной программной системы. В зависимости от специфики системы база пользователей может включать нетехнических или технических пользователей.

Собеседования по проектированию систем проводятся на большинство позиций в области программной инженерии и архитектур и на руководящие инженерные должности. (В этой книге мы будем называть сотрудников, занимающих такие позиции, общим словом *инженеры*.) Помимо этого, интервью включает оценку навыков программирования и поведенческих/культурных качеств кандидата.

1.1. ОБСУЖДЕНИЕ КОМПРОМИССОВ

Следующие факторы подтверждают важность собеседований по проектированию систем и подготовки к ним в качестве соискателя и эксперта.

Проверка эффективности соискателя на собеседованиях по проектированию систем используется для оценки широты и глубины его знаний в области проектирования систем, а также его умения выстроить общение и обсуждать дизайн систем с другими инженерами. Это критический фактор для определения уровня вашей будущей должности. Важность способности проектировать и анализировать крупномасштабные системы возрастает по мере продвижения к верхним уровням инженерной иерархии. Соответственно, собеседования по проектированию систем получают больший вес при приеме на руководящие должности. Не пожалейте времени на подготовку к ним в роли как эксперта, так и соискателя — если вы планируете карьеру в технологической области, это время окупится.

У технологической отрасли есть одна уникальная особенность: инженеры обычно меняют компанию каждые несколько лет, в отличие от других отраслей, где сотрудник может оставаться в компании много лет и даже на протяжении всей карьеры. Это значит, что типичный инженер проходит собеседования по проектированию систем не один раз. Инженеры, работающие в престижных компаниях, проводят еще больше собеседований по проектированию систем в качестве эксперта. У вас как у соискателя есть меньше часа на то, чтобы произвести лучшее возможное впечатление, а кандидаты, которые являются вашими конкурентами, принадлежат к числу самых умных и мотивированных людей в мире.

Проектирование систем — скорее искусство, чем наука. Его суть не в достижении совершенства, а в принятии компромиссного варианта дизайна, который можно реализовать с имеющимися ресурсами и временем и который будет наиболее близок к текущим и возможным будущим требованиям. Все обсуждения систем в этой книге подразумевают всевозможные оценки и предположения; они не являются академически строгими, полными или научными. В книге могут упоминаться паттерны проектирования и архитектуры, но мы не будем формально описывать их принципы. За дополнительной информацией читателям следует обращаться к другим источникам.

Цель собеседования по проектированию систем не получить правильные ответы, а выявить, насколько соискатель способен обсуждать разные возможные решения и оценивать их компромиссы в отношении соответствия требованиям. Знание разных типов требований и распространенных систем, встречающихся в части 1, поможет вам проектировать систему, оценить возможные подходы и обсудить их сильные и слабые стороны.

1.2. СТОИТ ЛИ ЧИТАТЬ ЭТУ КНИГУ?

Открытая природа собеседований по проектированию систем существенно усложняет подготовку к ним, поскольку невозможно заранее узнать, как и что будет обсуждаться во время собеседования. Инженер или студент, который ищет в интернете материалы по собеседованиям по проектированию систем, найдет огромный объем контента, который сильно отличается по качеству и разнообразию рассматриваемых тем. Все это создает путаницу и усложняет обучение. Более того, до недавнего момента почти не было книг, посвященных теме собеседований по проектированию систем, — как сказал знаменитый французский поэт и романист XIX века Виктор Гюго, «идее, время которой пришло». То, что одна мысль приходит в одно время сразу многим, подтверждает ее актуальность.

В книге предлагается структурированный и упорядоченный подход, который поможет начать подготовку к собеседованию по проектированию систем или заполнить пробелы в знаниях и понимании материала, возникающие при изучении большого объема разрозненных материалов. Что не менее важно, она научит вас, как продемонстрировать в ходе собеседования свою инженерную зрелость и навыки общения, в частности умение ясно и лаконично формулировать мысли и показывать знания, а также задавать вопросы эксперту за короткое время (примерно 50 минут).

Собеседование по проектированию систем, как и любое другое, также проверяет навыки общения, сообразительность, умение задавать правильные вопросы и преодолевать страх перед возможной неудачей. Вы можете забыть упомянуть что-то, что рассчитывает услышать эксперт. На тему о том, является ли такой формат собеседования ограниченным, можно спорить до бесконечности. По нашему опыту, с повышением уровня должности инженеры проводят все больше времени на совещаниях, и для них важнейшими навыками в том числе являются сообразительность, умение задавать правильные вопросы, направлять обсуждение на самые важные и актуальные темы и лаконично формулировать свои мысли. В этой книге подчеркивается, что соискатель должен эффективно и кратко обрисовать свой опыт проектирования систем на собеседовании, занимающем менее часа, и вести беседу в нужном направлении, задавая эксперту правильные вопросы. Чтение этой книги, наряду с отработкой проектирования систем с другими инженерами, позволит вам развить знания и красноречие, необходимые для прохождения собеседований, и эффективно участвовать в проектировании систем в компании, к которой вы присоединяетесь. Книга также пригодится экспертам, проводящим собеседования по проектированию систем.

Соискатель, у которого навыки письменного общения развиты сильнее навыков устного, может забыть упомянуть важные моменты в ходе примерно 50-минутного собеседования. На таких интервью преимущество имеют инженеры с хорошо

развитыми навыками устного общения, а у тех, у кого эти навыки слабы, шансы получить предложение о работе ниже, даже если они обладают значительным опытом проектирования систем и вносили ценный вклад в проектирование в организациях, в которых они работали. Эта книга подготовит инженеров к подобным (и не только) трудностям на собеседованиях по проектированию систем; покажет, как организованно решать их, и научит их не бояться.

Если вы — разработчик, желающий расширить свои знания концепций проектирования систем, усовершенствовать свои навыки их обсуждения или просто ознакомиться с их подборкой и примерами обсуждений, продолжайте читать.

1.3. ОБЗОР КНИГИ

Книга разделена на две части. Часть 1 напоминает учебник. В ее главах рассматриваются темы, поднимаемые на собеседованиях по проектированию систем. Часть 2 состоит из обсуждений типичных вопросов, относящихся к концепциям из части 1, которые задают на собеседованиях; в ней также рассматриваются антипаттерны, часто встречающиеся заблуждения и ошибки. В этих обсуждениях мы также констатируем очевидное: никто не ждет от соискателя полного знания во всех областях. Скорее он должен уметь обосновать, что некоторые подходы обеспечивают лучшее соответствие требованиям при допущении определенных компромиссов. Например, вам не придется вычислять степень сокращения размера файлов, затраты процессорного времени и памяти, необходимых для сжатия файлов в формате Gzip, но вы должны уметь объяснить, что сжатие файла перед его отправкой приведет к снижению сетевого трафика, хотя и потребует дополнительных затрат процессорного времени и памяти на стороне отправителя и получателя.

Книга старается собрать воедино разные актуальные материалы по теме. Это позволит вам сформировать базу своих знаний или же выявить пробелы в ней, чтобы позже заняться изучением других материалов.

Оставшаяся часть этой главы — своего рода пролог к примеру проектирования системы, в котором упомянуты некоторые концепции из части 1. Мы будем рассматривать эти концепции в посвященных им главах на основе данного контекста.

1.4. ПРОЛОГ: КОРОТКО О МАСШТАБИРОВАНИИ СЕРВИСОВ СИСТЕМЫ

Начнем с краткого описания типичной исходной конфигурации приложения и общего подхода к масштабированию сервисов приложения по мере необходимости. Попутно перечислим многие термины и концепции, а также виды

сервисов, необходимые для технологической компании, которые более подробно рассмотрим в оставшейся части книги.

ОПРЕДЕЛЕНИЕ Масштабируемостью (scalability) сервиса называется возможность простого и экономичного изменения ресурсов, выделенных сервису, для адаптации к изменениям нагрузки. Это понятие относится как к увеличению, так и к снижению количества пользователей и/или запросов к системе. Тема более подробно рассматривается в главе 3.

1.4.1. Начало: малое исходное развертывание приложения

На волне популярности домашней выпечки создадим привлекательное клиентское приложение Beigel[1], в котором пользователи могут читать и писать свои посты о ближайших кафе-пекарнях.

Изначально приложение Beigel включает следующие компоненты:

- Пользовательские приложения. По сути, это три версии одного приложения для трех основных платформ:
  - Браузерное приложение. Это приложение на базе ReactJS, которое отправляет запросы к сервису времени выполнения, написанному на JavaScript. Чтобы сократить объем кода JavaScript, загружаемого пользователями, мы сжимаем его при помощи Brotli. Gzip — более традиционный и популярный вариант, но Brotli создает сжатые файлы меньшего размера.
  - Приложение для iOS, загружаемое на iOS-устройство пользователя.
  - Приложение для Android, также загружаемое на Android-устройство пользователя.
- Бэкенд-сервис без сохранения состояния, обслуживающий пользовательские приложения. Может быть написан на Go или на Java.
- База данных SQL, размещенная на одном облачном хосте.

В приложении два основных сервиса: фронтенд и бэкенд. Эти компоненты показаны на рис. 1.1. Как видно из диаграммы, пользовательские приложения представляют собой компоненты на стороне клиента, а сервисы и базы данных — компоненты на стороне сервера.

ПРИМЕЧАНИЕ В разделах 6.5.1 и 6.5.2 объясняется, для чего нужен фронтенд-сервис между браузером и бэкендом.

---

[1] От *англ.* bagel — бейгл, рогалик (вид выпечки). — *Примеч. ред.*

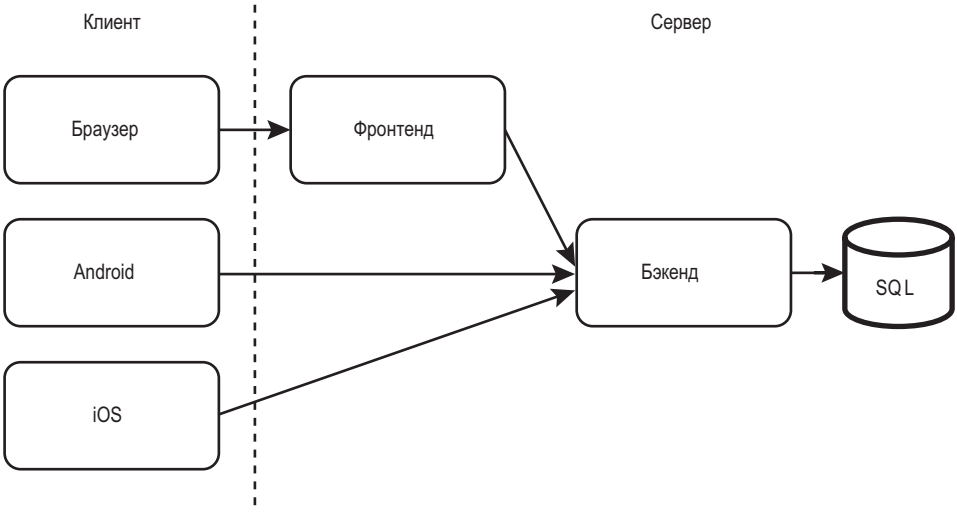

Рис. 1.1. Исходный дизайн системы приложения. Более подробное обоснование создания трех клиентских и двух серверных приложений (без учета приложения SQL/ базы данных) содержится в главе 6

При первом запуске сервиса количество пользователей будет небольшим и, как следствие, частота поступления запросов низкой. Для такой частоты может быть достаточно одного хоста. Мы настроим DNS для перенаправления всех запросов этому хосту.

Изначально два сервиса размещаются в одном датацентре, каждый на одном облачном хосте. (Облачный хостинг сравнивается с размещением на физическом оборудовании в следующем разделе.) Мы настраиваем DNS для перенаправления всех запросов из браузерного приложения на хост Node.js и от хоста Node.js и двух мобильных приложений на хост внутреннего сервиса.

1.4.2. Масштабирование с GeoDNS

Через несколько месяцев количество пользователей из Азии, Европы и Северной Америки, ежедневно проявляющих активность в Beigel, составляет сотни тысяч. В периоды пикового трафика фоновый сервис получает тысячи запросов в секунду, и мониторинговая система начинает отвечать кодом статуса 504 из-за тайм-аута. Система нуждается в масштабировании.

Мы своевременно заметили рост трафика и подготовились к нему. Наш сервис работает без сохранения состояния, как рекомендуют принятые лучшие практики, поэтому мы можем развернуть несколько одинаковых хостов бэкенда и разместить каждый хост в отдельном датацентре в своей части света. Обращаясь

к рис. 1.2, когда клиент выдает запрос к бэкенду через домен *beigel.com*, мы используем GeoDNS для направления клиента в ближайший к нему датацентр.

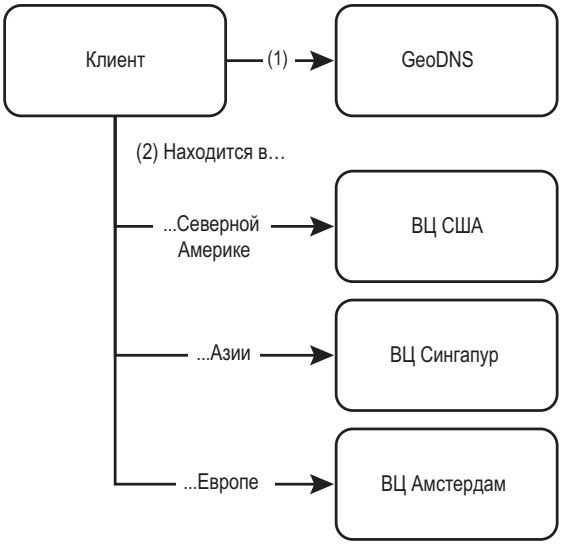

Рис. 1.2. Сервис может предоставляться в нескольких географически распределенных датацентрах. В зависимости от местонахождения клиента (определяемого по его IP-адресу) он получает IP-адрес хоста ближайшего центра данных, в который отправляет свои запросы. Клиент может кэшировать IP-адрес этого хоста

Если сервис обслуживает пользователей из определенной страны или географического региона в целом, он обычно размещается в близлежащем датацентре для минимизации задержки. Если же сервис обслуживает большую географически распределенную пользовательскую базу, его можно развернуть в нескольких датацентрах и использовать GeoDNS для возвращения пользователю IP-адреса сервиса, размещенного в ближайшем датацентре. Задача решается назначением нашему домену нескольких адресных записей для разных мест и IP-адреса по умолчанию для других мест. (*Адресная запись* — элемент конфигурации DNS, связывающий домен с IP-адресом.)

Когда клиент отправляет запрос серверу, GeoDNS получает информацию о местонахождении клиента по IP-адресу и назначает клиенту соответствующий IP-адрес хоста. В маловероятном, но возможном случае недоступности датацентра GeoDNS может вернуть IP-адрес сервиса в другом центре. Этот IP-адрес может кэшироваться на разных уровнях, включая провайдера (ISP) пользователя, уровни ОС и браузера.

1.4.3. Добавление сервиса кэширования

Согласно схеме на рис. 1.3, мы развертываем сервис кэширования Redis для обслуживания кэшированных запросов от пользовательских приложений. Мы выбрали несколько внутренних конечных точек с интенсивным трафиком, которые должны обслуживаться из кэша. Это дало немного времени на дальнейшую проработку системы, так как пользовательская база и нагрузка в виде запросов продолжили расти. Потребовались дополнительные усилия по масштабированию.

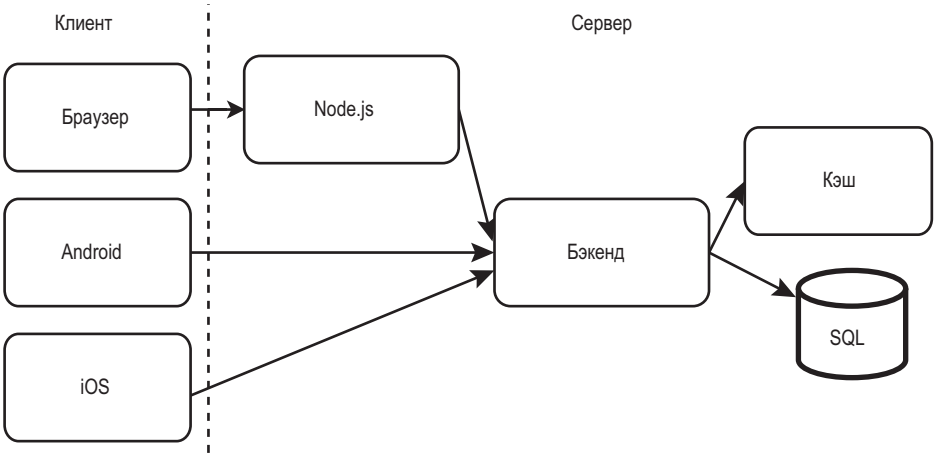

Рис. 1.3. Добавление кэширования к сервису. Некоторые внутренние конечные точки с интенсивным трафиком могут кэшироваться. Бэкенд запрашивает данные из БД при кэш-промахе или для баз данных/таблиц SQL, которые не были кэшированы

1.4.4. Сеть распространения контента

Наше приложение для браузера предоставляет статический контент/файлы, которые отображаются одинаково для всех пользователей и не зависят от пользовательского ввода: JavaScript, библиотеки CSS, графику и видео. Мы разместили эти файлы в репозитории исходного кода приложения, а пользователи загружали их из сервиса Node.js вместе с остальными частями приложения. Как показано на рис. 1.4, для размещения статичного контента было решено использовать стороннюю сеть распространения контента (CDN). Мы выбрали и обеспечили достаточный объем памяти в CDN для размещения файлов, отправили свои файлы в экземпляр CDN, переписали код для загрузки файлов по URL-адресам из CDN и удалили файлы из репозитория исходного кода.

На рис. 1.5 CDN хранит копии статических файлов в датацентрах по всему миру, чтобы пользователь загружал эти файлы из центра с наименьшей

задержкой — обычно самого близкого датацентра в географическом отношении, хотя другие датацентры могут оказаться быстрее, если ближайший центр находится под значительной нагрузкой или столкнулся с частичным сбоем.

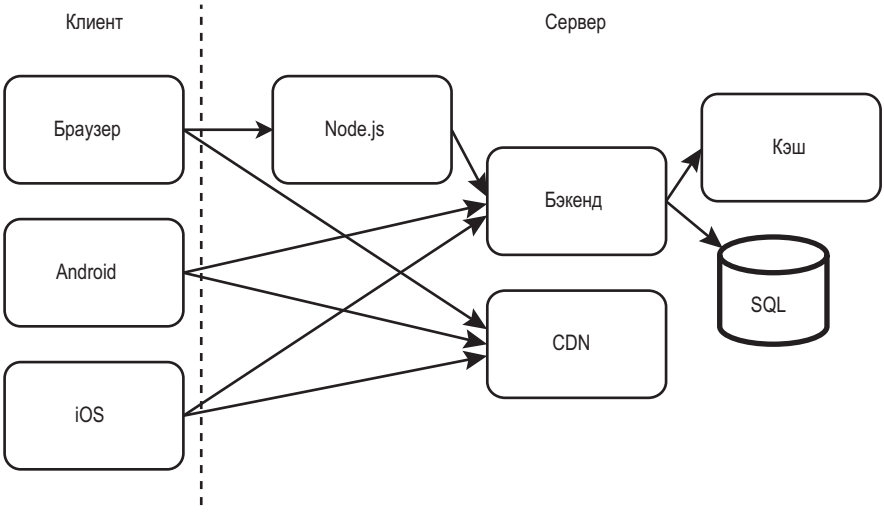

Рис. 1.4. Добавление CDN к сервису. Клиенты могут получать адреса CDN от бэкенда, или некоторые адреса CDN могут быть жестко закодированы в клиентах или сервисе Node.js

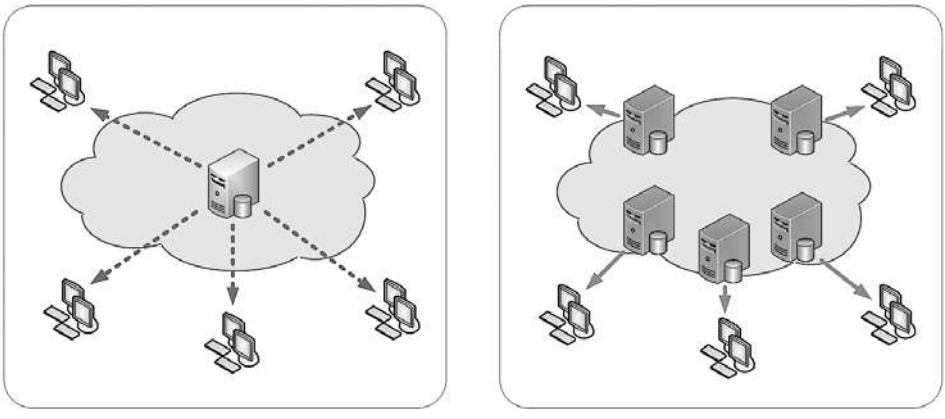

Рис. 1.5. На схеме слева все клиенты загружают данные с одного хоста. Справа клиенты загружают данные с разных хостов CDN. (Авторское право cc-by-sa https://creativecommons.org/licenses/by-sa/3.0/. Графика Kanoha, см. https://upload.wikimedia.org/wikipedia/commons/f/f9/NCDN_-_CDN.png)

Использование CDN снижает задержку и повышает пропускную способность, надежность и экономичность. (Все эти концепции рассматриваются в главе 3.)

При использовании CDN удельные затраты сокращаются, поскольку расходы на обслуживание, интеграцию и поддержку распределяются по большей нагрузке. Самые популярные CDN — CloudFlare, Rackspace и AWS CloudFront.

1.4.5. Коротко о горизонтальной масштабируемости и управлении кластерами, непрерывной интеграции и непрерывном развертывании

Наши сервисы — фронтенд и бэкенд — идемпотентны (некоторые преимущества идемпотентности описаны в разделах 4.6.1, 6.1.2 и 7.7); как следствие, они горизонтально масштабируемы, так что мы можем предоставить больше хостов для обработки увеличенной запросной нагрузки, не меняя исходный код, и развертывать фронтенд- и бэкенд-сервисы на этих хостах по мере надобности.

Над исходным кодом каждого из этих сервисов работают несколько инженеров. Инженеры ежедневно сохраняют новые коммиты. Мы изменяем существующие практики разработки и выпуска релизов на подходящие для поддержки большей команды и ускорения разработки, нанимая двух инженеров DevOps для разработки инфраструктуры управления большим кластером. Так как требования к масштабированию сервиса могут быстро измениться, необходима возможность легко изменять размеры кластера, развертывать сервисы и необходимые конфигурации на новых хостах и развертывать изменения кода на всех хостах кластера нашего сервиса. Мы можем экспериментировать с обширной пользовательской базой, развертывая разные версии кода или конфигурации на разных хостах. В этом разделе кратко рассматривается управление кластером для обеспечения горизонтального масштабирования и проведения экспериментов.

Непрерывная интеграция/непрерывное развертывание и инфраструктура как код

Чтобы выпускать релизы новой функциональности быстро и с минимальным риском ошибок, мы осуществляем непрерывную интеграцию и непрерывное развертывание (CI/CD) в Jenkins, а также средствами модульного и интеграционного тестирования. (Подробное обсуждение непрерывной интеграции/непрерывного развертывания выходит за рамки книги.) Docker используется для контейнеризации сервисов, Kubernetes (или Docker Swarm) — для управления кластером, включая масштабирование и балансировку нагрузки, а Ansible или Terraform — для управления конфигурацией разных сервисов, работающих на разных кластерах.

ПРИМЕЧАНИЕ Технология Mesos в целом считается устаревшей. Kubernetes — однозначный фаворит в этой области. Пара актуальных статей по теме: https://thenewstack.io/apache-mesos-narrowly-avoids-a-move-to-the-attic-for-now/ и https://www.datacenterknowledge.com/business/after-kubernetes-victory-its-former-rivals-change-tack.

Terraform позволяет инженеру по инфраструктуре создать единую конфигурацию, совместимую с разными облачными провайдерами. Конфигурация определяется на языке предметной области (DSL, Domain-Specific Language) Terraform и взаимодействует с облачными API для предоставления инфраструктуры. На практике конфигурация Terraform может содержать код, зависящий от поставщика; нужно постараться свести объем такого кода к минимуму. Как следствие, инфраструктура в меньшей степени привязана к конкретному поставщику.

Этот подход также известен под названием *«инфраструктура как код»*. Под этим термином понимается процесс управления и обеспечения работы датацентров с использованием машиночитаемых файлов определений вместо конфигурации физического оборудования или интерактивных средств настройки конфигурации (Wittig, Andreas; Wittig, Michael [2016]. Amazon Web Services in Action. Manning Publications. p. 93. ISBN 978-1-61729-288-0).

Постепенные развертывания и откаты

В этом разделе кратко рассматриваются постепенные развертывания и откаты, чтобы мы могли сравнить их с экспериментами в следующем разделе.

Развертывание сборки на продакшен можно проводить постепенно. Сборку можно развернуть на определенном проценте хостов, понаблюдать за их работой, а затем увеличить процент, постепенно доводя его до 100%. Например, можно последовательно развертывать сборку на 1, 5, 10, 25, 50, 75 и, наконец, 100% хостов. Развертывания можно вручную или автоматически откатить при обнаружении проблем:

- ошибок, пропущенных в ходе тестирования;
- сбоев;
- слишком высоких задержек или тайм-аутов;
- утечек памяти;
- повышенного потребления ресурсов (процессора, памяти, дискового пространства);
- увеличения оттока пользователей. Возможно, также стоит проанализировать постепенный отток пользователей — когда одни пользователи регистрируются и работают с приложением, а дргие перестают им пользоваться. Можно постепенно переводить все больший процент пользователей на новую сборку и изучать ее влияние на отток пользователей. Отток может возникать как из-за упомянутых факторов, так и из-за непредвиденных проблем, например из-за того, что изменения не понравились многим пользователям.

Например, в новой сборке задержка может превышать приемлемый уровень. Для решения этой проблемы можно воспользоваться комбинацией кэширования

и динамической маршрутизации. Сервис может заявлять задержку в 1 секунду. Когда клиент выдает запрос, направляемый к новой сборке, и происходит тайм-аут, клиент может читать результат из кэша или повторить запрос, который будет направлен хосту с более старой сборкой. Запросы и ответы следует сохранять в журнале, чтобы в дальнейшем провести диагностику тайм-аутов.

Можно настроить канал непрерывного развертывания, чтобы разбить рабочий кластер на несколько групп; инструментарий непрерывного развертывания будет определять подходящее количество хостов в каждой группе и закреплять хосты за разными группами. При увеличении размера кластера можно осуществлять переназначения и повторные развертывания.

Эксперименты

Когда вы вносите изменения в интерфейс при разработке новой функциональности (или ее удалении) или эстетические изменения в дизайн приложения, желательно развертывать их постепенно для группы пользователей, увеличивая размер этой группы, а не для всех пользователей сразу. Цель экспериментов — определение эффекта изменений в интерфейсе на поведение пользователя, в отличие от постепенного развертывания, направленного на изучение эффекта нового развертывания на эффективность приложения и отток пользователей. Стандартные методы проведения экспериментов — A/B-тестирование и многомерное тестирование (например, многорукий бандит). Эти темы выходят за рамки книги. За дополнительной информацией об A/B-тестировании обращайтесь по адресу https://www.optimizely.com/optimization-glossary/ab-testing/. Узнать подробнее о многомерном тестировании можно в книге «Experimentation for Engineers from A/B Testing to Bayesian Optimization»[1] Дэвида Свита (David Sweet) (Manning Publications, 2023), а метод многорукого бандита рассматривается в статье https://www.optimizely.com/ optimization-glossary/multi-armed-bandit/.

Эксперименты также проводятся для предоставления персонализированного опыта взаимодействия. Другое отличие экспериментов от постепенного развертывания/отката заключается в том, что в ходе экспериментов процент хостов, на которых выполняются разные сборки, часто оптимизируется при помощи средств включения/отключения функциональности, предназначенных для этой цели, тогда как при постепенном развертывании/откате механизм непрерывного развертывания используется для ручного или автоматического отката хостов к предыдущим сборкам при обнаружении проблем.

Непрерывное развертывание и эксперименты делают возможными короткие циклы обратной связи для новых развертываний и внедряемой функциональности.

---

[1] Свит Д. «Тюнинг систем: экспериментирование для инженеров от A/B-тестирования до байесовской оптимизации». СПб., издательство «Питер».

В веб-приложениях и приложениях бэкенда каждая версия функционала интерфейса (UX, user experience) обычно упаковывается в отдельную сборку. На части хостов будет работать другая сборка. С мобильными приложениями дело обычно обстоит иначе. Разные модификации UX часто кодируются в одной сборке, но каждый отдельный пользователь получает доступ только к определенному набору вариантов. Это делается по нескольким причинам:

- Развертывание мобильных приложений должно осуществляться через магазин приложений. Чтобы новая версия появилась на пользовательских устройствах, может понадобиться много часов. Быстрый откат развертывания невозможен.
- По сравнению с Wi-Fi мобильные данные медленнее, менее надежны и более дороги. Низкая скорость и ненадежность означают, что значительная часть контента должна предоставляться офлайн, то есть уже присутствовать в приложении. Тарифы на мобильную передачу данных во многих странах все еще высоки, при этом на нее могут устанавливаться лимиты и дополнительная плата за их превышение. Мы должны постараться оградить пользователей от этих затрат, в противном случае они будут реже использовать приложение, а то и вовсе удалят его. Чтобы проводить эксперименты и минимизировать использование данных загружаемых компонентов и медиа, мы просто включаем все эти компоненты и медиа в приложение и предоставляем доступ к нужному набору каждому конкретному пользователю.
- Мобильное приложение также может включать функциональность, которую часть пользователей никогда не будут применять. Например, в разделе 15.1 рассматриваются разные методы оплаты в приложениях. В мире существуют тысячи платежных решений. Приложение должно включать весь код и все SDK для всех решений, чтобы предоставить каждому пользователю небольшой набор платежных решений, доступных ему.

Как следствие, размер мобильного приложения вполне может превышать 100 Мбайт. Способы решения этой проблемы выходят за рамки книги; необходимо выдержать баланс, рассмотрев все компромиссы. Например, установка мобильного приложения YouTube очевидно не может включать множество видеороликов на YouTube.

1.4.6. Функциональная декомпозиция и централизация сквозных обязанностей

Целью функциональной декомпозиции является распределение разных функций по разным сервисам или хостам. Во многих сервисах присутствуют общие обязанности, которые могут выделяться в общие сервисы. В главе 6 обсуждаются причины, преимущества и компромиссы такого подхода.

Общие сервисы

Наша компания стремительно расширяется. Количество пользователей, ежедневно проявляющих активность в приложении, выросло до миллионов. Мы расширили команду до пяти инженеров iOS, пяти инженеров Android, 10 фронтенд-инженеров, 100 бэкенд-инженеров, а также создали группу по работе с данными.

Расширенная команда инженеров может работать над многими сервисами помимо приложений, непосредственно используемых клиентами, например сервисами для расширения поддержки пользователей и отдела эксплуатации. Для пользователей мы добавляем функциональность связи со службой поддержки, а для отдела эксплуатации — функциональность создания и запуска модификаций продуктов.

Во многих наших приложениях используется поле поиска. Мы создаем общий сервис поиска с использованием Elasticsearch.

Помимо горизонтального масштабирования, мы используем функциональную декомпозицию для распределения обработки данных и запросов по большому числу географически распределенных хостов, основанную на функциональности и географии. Мы уже применяли функциональную декомпозицию кэша, сервисов Node.js, бэкенда и базы данных по разным хостам; функциональная декомпозиция также применялась к другим сервисам при размещении каждого сервиса в собственном кластере географически распределенных хостов. На рис. 1.6 представлены общие сервисы, добавленные в Beigel.

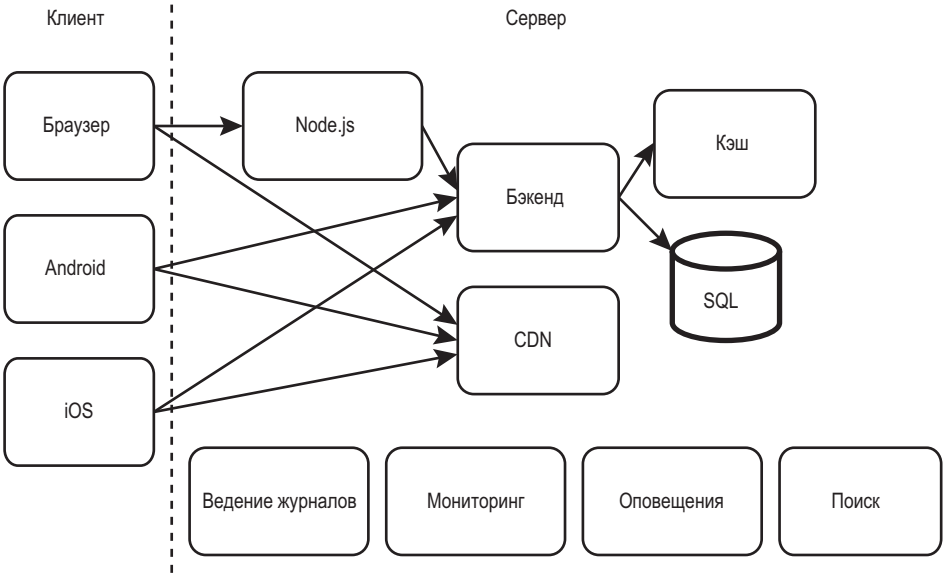

Рис. 1.6. Функциональная декомпозиция. Добавление общих сервисов

Мы добавили сервис ведения журналов, состоящий из брокера сообщений на базе журналов. Можно использовать Elastic Stack (Elasticsearch, Logstash, Kibana, Beats). Также используется распределенная система трассировки (такая, как Zipkin, Jaeger или распределенная система журналов) для трассировки запросов при обходе разных сервисов. Наши сервисы присоединяют идентификаторы каждому запросу, чтобы их можно было собрать как трассировочные данные и проанализировать. Ведение журналов, мониторинг и оповещения рассматриваются в разделе 2.5.

Также были добавлены сервисы мониторинга и оповещения. Мы строим внутренние браузерные приложения, чтобы сотрудники поддержки могли лучше помогать пользователям. Эти приложения обрабатывают журналы пользовательских приложений, генерируемые пользователем, и представляют их в удобном пользовательском интерфейсе, чтобы сотрудникам поддержки было проще понять проблему клиента.

Шлюз API и сервисная сеть (service mesh) — два основных способа централизации сквозных обязанностей. Другие возможные решения — паттерн «Декоратор» и аспектно-ориентированное программирование — выходят за рамки этой книги.

Шлюз API

К этому времени пользователи приложений составляют менее половины запросов API. Большинство запросов поступает от других компаний, которые предоставляют такие услуги, как рекомендация полезных продуктов и сервисов нашим пользователям на основании их активности в приложении. Слой шлюза API был разработан для того, чтобы предоставить доступ к части нашего API внешним разработчикам.

Шлюз API можно рассматривать как обратный прокси-сервер, маршрутизирующий запросы соответствующим бэкенд-сервисам. Он предоставляет общую функциональность, чтобы не дублировать ее в каждом отдельном сервисе:

- авторизации и аутентификации, а также другие политики управления доступом и безопасности;
- ведения журналов, мониторинга и оповещения на уровне запросов;
- ограничения частоты;
- выставления счетов;
- аналитики.

Наша исходная архитектура, в которой задействован шлюз API и его сервисы, изображена на рис. 1.7. Запрос к сервису проходит через централизованный шлюз API. Шлюз API предоставляет всю функциональность, описанную выше, проводит преобразование DNS, а затем перенаправляет запрос хосту соответствующего сервиса. Шлюз API выдает запросы к таким сервисам, как DNS, система управления идентификацией и доступом, сервис конфигурации ограничения

частоты и т. д. Также через шлюз API регистрируются все изменения, вносимые в конфигурацию.

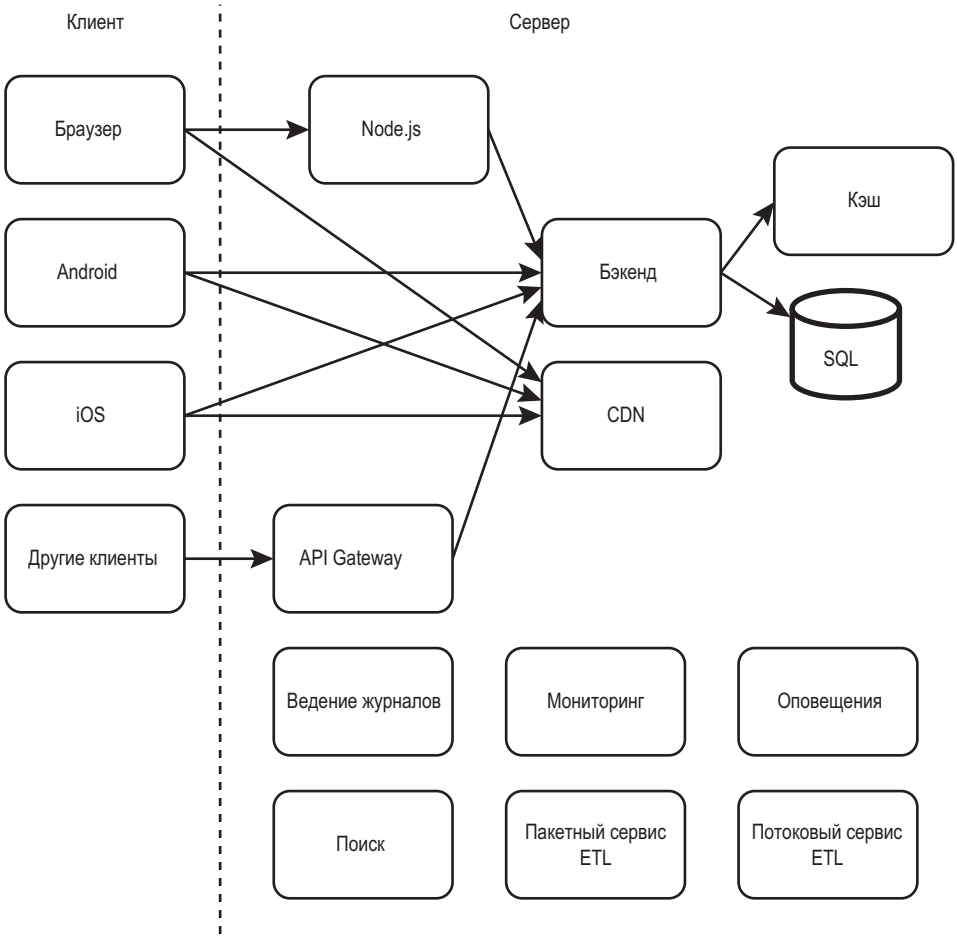

Рис. 1.7. Исходная архитектура со шлюзом API и сервисами. Запросы к сервисам проходят через шлюз API

Тем не менее у этой архитектуры имеется ряд недостатков. Шлюз API создает дополнительную задержку и требует большого кластера хостов. Хост шлюза API и хост сервиса, обслуживающего конкретный запрос, могут находиться в разных датацентрах. Дизайн системы, который пытается маршрутизировать запросы через хосты шлюза API и хосты сервисов, будет громоздким и излишне сложным.

Проблема решается использованием сервисной сети, также называемой паттерном Sidecar. Концепция сервисной сети более подробно рассматривается в главе 6.

Схема сети изображена на рис. 1.8. Для ее создания можно воспользоваться специализированным фреймворком, таким как Istio. Каждый основной сервис, развернутый на отдельном хосте, может иметь дополнительный sidecar-контейнер. Для решения этой задачи мы используем модули (pods) Kubernetes. Каждый модуль может содержать свой сервис (в одном контейнере), а также sidecar-контейнер.

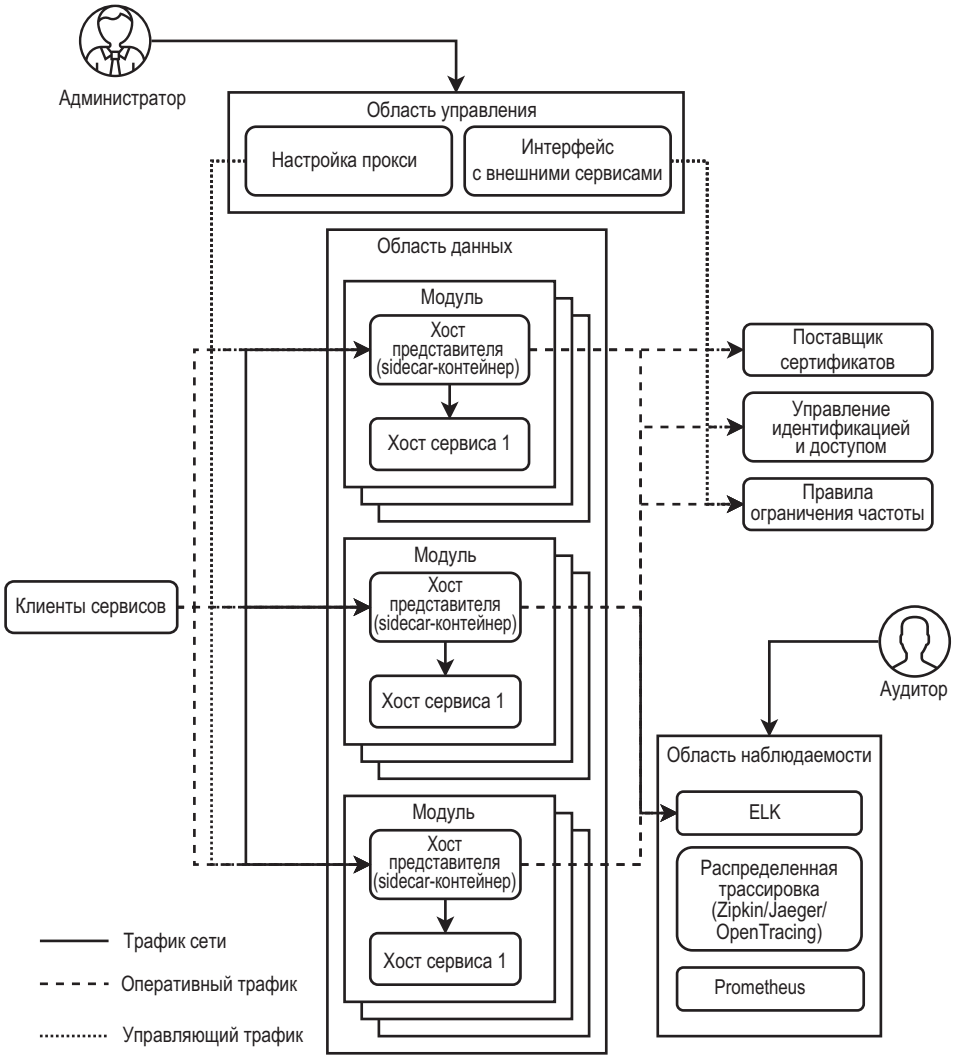

Рис. 1.8. Сервисная сеть. Prometheus выдает запросы к каждому прокси-хосту для получения / автоматизированного сбора метрик, но для простоты на диаграмме эти взаимодействия не показаны. Иллюстрация позаимствована с https://livebook.manning.com/book/cloud-native/chapter-10/146

Мы предоставляем административный интерфейс для настройки политик, и их конфигурации могут распространяться и на sidecar-контейнеры сервиса.

В такой архитектуре все запросы и ответы сервиса маршрутизируются через sidecar-контейнер. Сервис и sidecar размещаются на одном хосте (то есть на одной машине), чтобы они могли обращаться друг к другу через локальный хост, без сетевой задержки. С другой стороны, sidecar-контейнер потребляет системные ресурсы.

Сервисная сеть без расширений — передовой рубеж технологий

Сервисная сеть требовала удвоения количества контейнеров в системе. Для систем, подразумевающих коммуникации между внутренними сервисами, можно сократить сложность за счет размещения логики посредника sidecar-контейнера в клиентские хосты, выдающие запросы к хостам сервисов. В структуре сервисной сети без sidecar-контейнеров клиентские хосты получают конфигурации из области управления. Клиентские хосты должны поддерживать API области управления, поэтому они также должны включать соответствующие библиотеки сетевых коммуникаций.

Недостаток сервисной сети без расширений заключается в том, что в схеме должен присутствовать клиент, написанный на том же языке, что и сервис.

Платформы сервисной сети без расширений все еще находятся на ранней стадии развития. Google Cloud Platform (GCP) Trafic Director — реализация, выпущенная в апреле 2019 года (https://cloud.google.com/blog/products/networking/trafic-director-global-trafic-management-for-open-service-mesh).

Разделение ответственности на команды и запросы (CQRS)

Разделение ответственности на команды и запросы (CQRS, Command Query Responsibility Segregation) — паттерн микросервисов, в котором операции команд/записи и запросов/чтения функционально разделены по разным сервисам. В частности, брокеры сообщений и задания ETL являются примерами CQRS. Любой дизайн, в котором данные записываются в одну таблицу, а затем преобразуются и вставляются в другую таблицу, также является примером CQRS. CQRS добавляет сложность, но имеет более низкую задержку и улучшенную масштабируемость, и его проще обслуживать и использовать. Сервисы записи и чтения могут масштабироваться по отдельности.

В книге встречается множество примеров CQRS, хотя мы не будем отдельно привлекать к ним внимание. Один из таких примеров приведен в главе 15, где хост Airbnb выполняет запись в сервис объявлений, а гости читают информацию из сервиса бронирования. (Хотя сервис резервирования также предоставляет гостям конечные точки записи для бронирования жилья, не связанные с хостом, обновляющим объявления.)

Более подробное определение CQRS легко найти в других источниках.

1.4.7. Пакетные и потоковые сервисы извлечения, преобразования и загрузки данных (ETL)

В части систем возникают непредсказуемые пики трафика, и некоторые запросы обработки данных не обязаны быть синхронными (то есть немедленно обрабатывать запрос и возвращать ответ). Вот несколько примеров.

- Некоторые запросы, требующие выборки больших объемов данных из баз данных (например, обрабатывающие гигабайты данных).
- Некоторые данные эффективнее периодически обрабатывать заранее, не только при поступлении запроса. Например, на домашней странице приложения могут выводиться 10 самых популярных изученных слов по всем пользователям за последний час или неделю. Эта информация должна обрабатываться заранее каждый час или ежедневно. Более того, результат этой обработки может повторно использоваться для всех пользователей, чтобы не повторять обработку для каждого пользователя.
- Иногда допустимо показывать данные, устаревшие на несколько часов или дней. Например, пользователям не обязательно видеть самую свежую статистику по количеству просмотров их контента. Вывод статистики, устаревшей на несколько часов, может стать вполне приемлемым решением.
- Операции записи (например, запросы к базам данных `INSERT`/`UPDATE`/`DELETE`), которые не обязательно выполнять немедленно. Предположим, запись в сервис ведения журналов не обязательно сразу же переносить на жесткие диски хостов сервиса. Такие запросы можно поместить в очередь и выполнить позже.

Если не использовать асинхронный подход (например, ETL) для таких систем, как ведение журналов, получающих большие объемы запросов от множества других систем, кластер системы ведения журналов должен будет содержать тысячи хостов, чтобы осуществлять синхронную обработку всех этих запросов.

Для подобных пакетных заданий можно использовать комбинацию событийных потоковых систем, таких как Kafka (или Kinesis, если использовать AWS), с пакетными инструментами ETL, такими как Airflow.

Если вам нужно непрерывно обрабатывать данные вместо периодического выполнения пакетных заданий, можно использовать такие потоковые средства, как Flink. Например, если пользователь вводит некие данные в приложении и вы хотите использовать эти данные для отправки ему рекомендаций или оповещений в течение секунд или минут, можно создать пайплайн Flink, обрабатывающий недавний ввод пользователя. Система ведения журналов обычно работает по потоковой схеме, потому что она ожидает видеть бесконечный

поток запросов. Если запросы происходят реже, пакетного пайплайна будет достаточно.

1.4.8. Другие распространенные сервисы

Со временем компания растет, а пользовательская база расширяется. Мы начинаем разрабатывать новые продукты, и эти продукты поддерживают все больше возможностей настройки и персонализации для обеспечения интересов огромной, растущей и разнообразной пользовательской базы. Чтобы удовлетворять новые требования, которые сопровождают этот рост, и извлечь из него практическую пользу, нам понадобится множество новых сервисов, в числе которых:

- Управление идентификационными данными клиентов/внешних пользователей для аутентификации/авторизации внешних пользователей.
- Разные сервисы хранения данных, включая сервисы баз данных. Специфические требования каждой системы означают, что существуют некоторые оптимальные способы долгосрочного хранения, обработки и предоставления используемых ею данных. Необходимо разрабатывать и обслуживать общие сервисы хранения, использующие разные технологии и методы.
- Асинхронная обработка. Обширная пользовательская база данных требует большего количества хостов и может создавать непредсказуемые выбросы трафика к сервисам. Чтобы справиться с пиковым трафиком, необходимо применить асинхронную обработку для эффективного использования оборудования и сокращения лишних затрат на него.
- Сервис аналитики и машинного обучения, включая эксперименты, создание моделей и развертывание. Большая пользовательская база может использоваться для экспериментов по определению предпочтений пользователей, персонализации опыта взаимодействия, привлечения новых пользователей и выявлению других способов повышения дохода.
- Внутренний поиск и подзадачи (например, сервис автозаполнения/опережающего ввода). Многие веб- и мобильные приложения могут иметь панели поиска для интересующих данных.
- Сервисы и группы обеспечения конфиденциальности. Растущая численность пользователей и большой объем клиентских данных привлекут злоумышленников (как внешних, так и внутренних), которые попытаются похитить данные. Незаконное использование большой пользовательской базы данных отразится на множестве людей и организаций. Вы должны позаботиться о защите конфиденциальности пользователя.
- Обнаружение попыток мошенничества. Рост доходов компании сделает ее соблазнительной целью для преступников и мошенников, так что наличие эффективной системы обнаружения попыток мошенничества обязательно.

1.4.9. Облачный хостинг и физическое оборудование

Вы можете управлять своими хостами и датацентрами самостоятельно или же доверить эти обязанности провайдерам облачных сервисов. В этом разделе приведен сравнительный анализ обоих решений.

Общие соображения

В начале этого раздела мы решили использовать облачные сервисы (с арендой хостов у таких провайдеров, как Amazon AWS, DigitalOcean или Microsoft Azure) вместо физического оборудования (когда вы владеете и управляете своими физическими машинами).

Облачные провайдеры предоставляют многие из нужных сервисов, включая непрерывную интеграцию/непрерывное развертывание (CI/CD), ведение журналов, мониторинг, оповещения, упрощенную настройку и управление разными типами баз данных, включая кэши, SQL и NoSQL.

Если изначально выбрать решение с физическим оборудованием, вам придется создавать и сопровождать все необходимые сервисы. Это может отвлечь внимание и время от разработки функциональности и дорого обойтись вашей компании.

Также необходимо учитывать затраты на труд инженеров в сравнении с затратами на облачные средства. Инженеры — очень дорогостоящий ресурс, и кроме денег хорошие инженеры обычно предпочитают творческие задачи. Если вы будете утомлять их такой рутиной, как мелкомасштабные конфигурации стандартных сервисов, они могут уйти в другую компанию, и найти им замену на конкурентном рынке будет непросто.

Облачные средства часто обходятся дешевле найма инженеров для настройки и обслуживания инфраструктуры с физическим оборудованием. Скорее всего, вы не сможете обеспечить экономию за счет масштаба и сопутствующую эффективность удельных затрат и не будете владеть специфическими компетенциями в отличие от облачных провайдеров. Если ваша компания успешна, она может достичь стадии развития, на которой экономия за счет масштаба позволит рассмотреть вариант с физическим оборудованием.

Использование облачных сервисов также имеет следующие преимущества.

Простота настройки

В браузерном приложении облачного провайдера легко выбрать пакет, наиболее подходящий для наших целей. На физическом оборудовании необходимо будет выполнить такие операции, как установка серверного ПО (например, Apache) или настройка сетевых соединений и переадресация портов.

Меньше расходов

Облачный хостинг не требует первоначальных затрат на приобретение физических машин/серверов. Облачный провайдер позволяет применять схему оплаты с увеличением стоимости по мере использования, а также предоставляет скидки за объем. Восходящее или нисходящее масштабирование в ответ на непредсказуемо изменяющиеся требования выполняется легко и быстро. Если выбрать физическое оборудование, может возникнуть ситуация, в которой физических машин оказывается слишком мало или слишком много. Кроме того, часть облачных провайдеров предлагают сервисы «автомасштабирования», которые автоматически изменяют размер кластера в соответствии с текущей нагрузкой.

Тем не менее облачные решения не всегда дешевле решений с физическим оборудованием. Dropbox (https://www.geekwire.com/2018/dropbox-saved-almost-75-million-two-years-building-tech-infrastructure/) и Uber (https://www.datacenterknowledge.com/uber/want-build-data-centers-uber-follow-simple-recipe) — два примера компаний, использующих хостинг в собственных датацентрах, потому что с их требованиями этот вариант был более экономичным.

Улучшенная поддержка и качество

Облачные сервисы на практике обычно обеспечивают более высокую производительность, лучшее взаимодействие и поддержку, с меньшим количеством серьезных сбоев. Возможно, это обусловлено тем, что облачные сервисы должны быть конкурентоспособными на рынке, чтобы привлекать и удерживать клиентов, по сравнению с физическим оборудованием, которое пользователи организации вынуждены использовать, не имея особых альтернатив. Многие организации склонны ценить и уделять больше внимания клиентам, нежели внутренним пользователям или сотрудникам, возможно потому, что доход от клиентов можно измерить напрямую, тогда как оценить объективно преимущества предоставления высококачественных сервисов и поддержки внутренним пользователям оказывается труднее. Как следствие, потери дохода и репутации от низкокачественных внутренних сервисов также трудно выразить количественно. Облачные сервисы также могут обеспечивать экономию за счет объема, в отличие от решений с физическим оборудованием, поскольку усилия команды облачного сервиса распределяются по большей пользовательской базе.

Документация для внешних пользователей может быть качественнее, чем внутренняя. Она может быть лучше написана, чаще обновляться и размещаться на хорошо организованном веб-сайте с удобными средствами поиска. На нее может выделяться больше ресурсов, что позволяет составить видеоматериалы и пошаговые руководства.

Внешние сервисы могут предоставлять более качественную проверку ввода по сравнению с внутренними сервисами. Возьмем простой пример: если некоторое

поле UI или поле точки API предназначено для ввода адреса электронной почты, сервис должен проверить, представляет ли ввод пользователя реальный адрес электронной почты. Компания должна уделять больше внимания внешним пользователям, жалующимся на низкое качество проверки ввода, потому что внешние пользователи могут отказаться от использования и оплаты продукта компании. Аналогичная обратная связь от внутренних пользователей, у которых нет особого выбора, может быть просто проигнорирована.

При возникновении ошибки качественный сервис должен возвращать содержательные сообщения, указывающие пользователю, как исправить ошибку (желательно без длительного общения с поддержкой или разработчиками сервиса). Внешние сервисы могут предоставлять более качественные сообщения об ошибках, а также выделять больше ресурсов для обеспечения качественной поддержки.

Если клиент отправляет сообщение, он может получить ответ в течение минут или часов, тогда как обработка вопроса сотрудника может занять часы или дни. Иногда вопросы по внутреннему каналу вообще остаются без ответа или же сотрудника отсылают к плохо написанной документации.

Внутренние сервисы организации в лучшем случае не уступают внешним сервисам, если организация выделяет для них достаточные ресурсы и стимулы. Так как улучшение опыта взаимодействия с пользователем и поддержки повышает энтузиазм и производительность пользователей, организация может подумать об определении метрик, оценивающих качество обслуживания внутренних пользователей. Один из способов избежать всех этих трудностей — использовать облачные сервисы. Эти соображения можно обобщить для сравнения внешних и внутренних сервисов.

Наконец, обязанностью каждого отдельного разработчика является соблюдение высоких стандартов в работе, а не предположения о качестве работы других. Тем не менее хронически плохое качество внутренних зависимостей может отрицательно сказываться на производительности и настроении в организации.

Обновления

Как аппаратные, так и программные технологии, используемые в физической инфраструктуре организации, со временем устаревают, а их обновление затрудняется. Это очевидно для финансовых компаний, использующих мейнфреймы. Переключение с мейнфреймов на серийные серверы — дело в высшей степени затратное, сложное и рискованное, так что такие компании продолжают покупать новые мейнфреймы, которые стоят намного дороже эквивалентных вычислительных мощностей серийных серверов. Организациям, использующим серийные серверы, также потребуются соответствующий опыт и усилия для постоянного обновления аппаратного и программного обеспечения. Например, даже обновление версии MySQL, используемой в крупной организации, потребует значительного времени и усилий. Многие организации предпочитают передавать обслуживание на аутсорс облачным провайдерам.

Недостатки

Одним из недостатков облачных провайдеров является зависимость от поставщика услуг. Для переноса данных и сервисов от одного облачного провайдера к другому могут потребоваться значительные технические усилия, а также оплата дублирования сервиса на время переноса.

Менять провайдера можно по разным причинам. Провайдер может быть эффективно управляемой компанией, которая успешно выполняет соглашение об уровне обслуживания (SLA, Service Level Agreement) с высокими требованиями, но нет никаких гарантий, что так будет всегда. Качество сервиса компании может ухудшиться в будущем, и провайдер нарушит условия SLA. Цена может стать неконкурентоспособной, так как физическое оборудование станет более дешевым или другие облачные провайдеры могут предложить более низкую ставку. А может, провайдер не сможет обеспечить необходимый уровень безопасности или другие требуемые характеристики.

Другим недостатком может быть слабый контроль конфиденциальности и безопасности данных и сервисов клиента. Например, вы не уверены, что облачный провайдер защитит ваши данные или обеспечит безопасность ваших сервисов. С физическим оборудованием можно лично проконтролировать конфиденциальность и безопасность.

По этим причинам многие компании принимают многооблачную стратегию с использованием нескольких облачных провайдеров вместо одного, чтобы можно было за минимальное время уйти от конкретного провайдера, если такая необходимость вдруг возникнет.

1.4.10. Бессерверная обработка: FaaS

Если некоторая конечная точка или функция используется редко или не имеет жестких требований к задержке, может быть дешевле реализовать ее как функцию на платформе FaaS (Function as a Service, «функция как сервис»), такой как AWS Lambda или Azure Functions. Выполнение функции только тогда, когда требуется, означает, что хостам не придется непрерывно ожидать запросов к этой функции.

OpenFaaS и Knative — решения FaaS с открытым исходным кодом, которые могут использоваться для поддержки FaaS на вашем собственном кластере или в виде прослойки на AWS Lambda для улучшения портируемости функций между облачными платформами. На момент написания книги интеграции между решениями FaaS с открытым кодом и другими FaaS, управляемыми провайдером, такими как Azure Functions, еще не существовало.

У функций Lambda установлен 15-минутный тайм-аут. Предполагается, что система FaaS будет обрабатывать запросы, которые завершаются за это время.

В типичной конфигурации сервис шлюза API получает входящие запросы и инициирует выполнение соответствующих функций FaaS. Использование шлюза API обусловлено необходимостью наличия непрерывно работающего сервиса, ожидающего запросов.

Другое преимущество FaaS заключается в том, что разработчикам сервисов не нужно управлять развертыванием и масштабированием и они могут сосредоточиться на разработке бизнес-логики.

Заметим, что для одного выполнения функции FaaS требуются такие шаги, как запуск контейнера Docker, запуск соответствующей исполнительной среды языка (Java, Python, Node.js и т. д.), запуск функции и завершение работы исполнительной среды и контейнера Docker. Обычно это называется холодным запуском (cold start). Фреймворки, запуск которых занимает несколько минут (как некоторые фреймворки Java), могут оказаться неподходящими для FaaS. Это стимулировало разработку JDK с быстрым запуском и малыми затратами памяти, таких как GraalVM (https://www.graalvm.org/).

Зачем все эти накладные расходы? Почему нельзя упаковать все функции в один пакет и выполнять его на всех экземплярах хостов, по аналогии с монолитной архитектурой? Это объясняется недостатками монолитов (см. приложение А).

Почему нельзя развернуть часто используемые функции на определенных хостах на определенное время (например, с ограниченным сроком действия)? Такая система напоминает автоматически масштабируемые микросервисы, и ее стоит рассмотреть при использовании фреймворков, запуск которых занимает много времени.

Портируемость FaaS — неоднозначная тема. На первый взгляд организация, которая активно использует такую закрытую FaaS, как AWS Lambda, привязывается к конкретному провайдеру; миграция на другое решение становится трудным, долгим и недешевым делом. Платформы FaaS с открытым исходным кодом не являются полноценной альтернативой, поскольку компания должна предоставлять собственные хосты и заниматься их обслуживанием, что противоречит цели масштабируемости FaaS. Проблема становится особенно серьезной в масштабных архитектурах, где FaaS может обходиться намного дороже физического оборудования.

Однако функция в FaaS может быть написана на двух уровнях: внутренней функции, содержащей основную логику, которая упаковывается во внешнюю функцию с конфигурациями для конкретных провайдеров. Чтобы переключить провайдера для любой внутренней функции, достаточно изменить только внешнюю функцию.

Spring Cloud Function (https://spring.io/projects/spring-cloud-function) — набирающий популярность фреймворк FaaS, обобщающий эту концепцию. Он поддерживается AWS Lambda, Azure Functions, Google Cloud Functions, Alibaba Function Compute и может поддерживаться другими провайдерами FaaS в будущем.

1.4.11. Заключение: масштабирование бэкенд-сервисов

В остальных главах части 1 обсуждаются концепции и методы масштабирования бэкенд-сервисов. Фронтенд/UI-сервисом обычно служит сервис Node.js, единственная задача которого — предоставлять любому пользователю браузерное приложение, написанное во фреймворке JavaScript (таком, как ReactJS или Vue.js), так что он может масштабироваться простой регулировкой размера кластера и использованием GeoDNS. Бэкенд-сервис динамичен и может возвращать разные ответы для каждого запроса. Методы его масштабирования более разнообразны и сложны. Функциональная декомпозиция рассматривалась в предыдущем примере, и мы еще будем возвращаться к ней по мере надобности.

ИТОГИ

- Подготовка к собеседованию по проектированию систем чрезвычайно важна для вашей карьеры; кроме того, она принесет пользу вашей компании.
- Собеседование по проектированию систем представляет собой обсуждение процесса проектирования программной системы, обычно предоставляемой по сети.
- GeoDNS, кэширование и CDN — основные методы масштабирования сервисов.
- Инструменты и практики непрерывной интеграции / непрерывного развертывания (CI/CD) позволяют ускорять выпуск релизов функциональности и снизить количество ошибок. Они также позволяют разделить пользователей на группы и предоставить каждой группе отдельную версию приложения для целей экспериментов.
- Инструменты типа «инфраструктура как код» (такие, как Terraform) представляют собой полезные средства автоматизации для управления кластерами, масштабирования и экспериментов с функциональностью.
- Функциональная декомпозиция и централизация сквозной функциональности — ключевые элементы проектирования систем.
- Задания ETL могут использоваться для распределения обработки выбросов трафика во времени, что сокращает требуемый размер кластера.
- Облачный хостинг обладает рядом преимуществ. Снижение расходов часто является одним из них, но не всегда. Среди недостатков можно выделить привязку к конкретному провайдеру и потенциальные риски для конфиденциальности и безопасности.
- Бессерверные системы — альтернативный подход к сервисам. Он более экономичный, так как хостам не приходится работать непрерывно, но при этом ограничивает функциональность.

2 Типичный ход собеседования по проектированию систем

В ЭТОЙ ГЛАВЕ

- ✓ Уточнение требований к системам и оптимизация возможных компромиссов
- ✓ Предварительное планирование спецификации API системы
- ✓ Проектирование моделей данных системы
- ✓ Обсуждение таких аспектов, как ведение журналов, мониторинг, оповещение и поиск
- ✓ Анализ прошедшего собеседования

В этой главе рассматриваются принципы, которые необходимо соблюдать во время собеседования по проектированию систем. Вернитесь к этому списку, дочитав книгу до конца. Придерживайтесь этих принципов во время интервью.

1. Уточните функциональные и нефункциональные требования (см. главу 3), например, в отношении QPS (количество запросов в секунду, Queries Per Second) и задержки 99-го процентиля. Спросите, удобно ли эксперту начать обсуждение с простой системы, а затем перейти к масштабированию и добавлению функциональности или следует сразу начать с проектирования масштабируемой системы.

2. Любое решение является компромиссным. У систем нет практически ни одной характеристики, у которой не было бы недостатков и которая не требовала бы компромиссов. Любое улучшение масштабируемости и целостности или уменьшение задержки также повышает сложность, затраты и требует внимания к безопасности, ведения журналов, мониторинга и оповещений.
3. Направляйте ход интервью. Поддерживайте интерес эксперта. Обсудите, что его интересует. Предлагайте ему интересующие его темы.
4. Помните о времени. За 1 час вам предстоит много всего обсудить.
5. Обсудите темы ведения журналов, мониторинга, оповещений и аудита.
6. Обсудите темы тестирования и обслуживания, включая удобство отладки, сложность, безопасность и конфиденциальность.
7. Продумайте и обсудите корректное снижение функциональности и возникновение отказов в системе и каждом ее компоненте, включая отказы без видимых признаков и замаскированные отказы. Ошибки могут никак не проявляться внешне. Никогда ничему не доверяйте. Не доверяйте внутренним или внешним системам. Не доверяйте своей собственной системе.
8. Рисуйте диаграммы, блок-схемы и диаграммы последовательности действий. Используйте их как визуальные вспомогательные материалы.
9. Систему всегда можно улучшить. Всегда можно обсудить что-то еще.

Обсуждение любого вопроса может продолжаться часами. Вам необходимо сосредоточиться на определенных аспектах, предложив эксперту разные направления для обсуждения и спросив, в каком из них следует двигаться. У вас меньше часа на то, чтобы показать полный объем ваших знаний. Вы должны обладать способностью анализировать и оценивать детали и плавно переходить к обсуждению более общих вопросов, таких как высокоуровневая архитектура и взаимозависимости, и, наоборот, низкоуровневых подробностей реализации всех компонентов. Если вы что-то забудете или решите не упоминать, эксперт предположит, что вы этого не знаете. Тренируйтесь обсуждать проектирование систем с коллегами-инженерами, чтобы совершенствоваться в этом искусстве. Престижные компании проводят собеседования с множеством хорошо подготовленных кандидатов, и каждый, кто прошел такое собеседование, свободно говорит на языке проектирования систем.

В этом разделе демонстрируются разные подходы к обсуждению тем собеседования. Многие из этих тем встречаются часто, поэтому повторения неизбежны. Используйте принятую в отрасли терминологию и старайтесь, чтобы каждая произнесенная вами фраза была информативной, ведь вы ограничены во времени.

Вот примерный ход беседы. Обсуждение проектирования систем весьма динамично и не обязательно будет проходить в указанном порядке:

1. Уточнение требований и обсуждение компромиссов.
2. Предварительная спецификация API.
3. Проектирование модели данных. Обсуждение возможной аналитики.
4. Обсуждение обработки отказов, корректного снижения функциональности, мониторинга и оповещений. Другие темы — узкие места, балансировка нагрузки, исключение единых точек сбоев, высокая доступность, восстановление после сбоев и кэширование.
5. Обсуждение сложностей и компромиссов, процессы обслуживания и вывода из эксплуатации, затраты.

2.1. УТОЧНЕНИЕ ТРЕБОВАНИЙ И ОБСУЖДЕНИЕ КОМПРОМИССОВ

Уточнение требований — первая «галочка», которую необходимо поставить в ходе собеседования. В главе 3 описаны подробности и важность обсуждения функциональных и нефункциональных требований.

В конце этой главы будет дано общее руководство по обсуждению требований в ходе собеседования. Мы будем повторять его для каждого вопроса части 2. Не забывайте о том, что каждое собеседование уникально, и вы можете отклоняться от этого руководства так, как того потребует ситуация.

Обсуждение функциональных требований не должно занимать более 10 минут, потому что это уже ≥ 20 % времени собеседования. Тем не менее внимание к деталям исключительно важно. Не записывайте функциональные требования по одному, обсуждая их по ходу дела, — так можно упустить какие-то из них. Вместо этого проведите быстрый мозговой штурм, составьте список функциональных требований, а потом переходите к обсуждению. Можно сказать эксперту, что вы хотите убедиться, что записали все критические требования, но помните об ограниченном времени.

Можно начать с 30-секундного или 1-минутного обсуждения общего назначения системы и ее места в картине бизнес-требований. Можно кратко упомянуть конечные точки, общие почти для всех систем: конечные точки контроля исправности, регистрации и входа. Все, о чем нужно говорить подробно, вряд ли будет упомянуто на собеседовании. Рассмотрим некоторые распространенные функциональные требования.

1. Категории/роли пользователей:
   - Кто и как будет пользоваться системой? Обсудите и запишите истории пользователей. Рассмотрите разные комбинации пользовательских

категорий, например, вручную/программно или потребительские/корпоративные. Например, комбинация «вручную/потребительские» подразумевает запросы от потребителей через мобильные или браузерные приложения. Комбинация «программно/корпоративные» включает запросы от других сервисов или компаний.

- Технические или нетехнические? Спроектируйте платформы или сервисы для разработчиков или для неспециалистов. Технические примеры — сервисы баз данных, такие как хранилища данных «ключ — значение», либо библиотеки для таких функций, как согласованное хеширование или сервисы аналитики. Нетехнические вопросы обычно формулируются в виде «Спроектируйте это известное потребительское приложение». В таких случаях следует обсуждать все категории пользователей, а не только нетехнических потребителей приложения.
- Перечислите пользовательские роли (например, покупатель, продавец, автор публикаций, читатель, разработчик, менеджер).
- Обращайте внимание на числа. Каждое функциональное и нефункциональное требование должно быть связано с числом. Выборка новостей? Сколько новостей? За какое время? За сколько миллисекунд/секунд/часов/дней?
- Предусмотрена ли коммуникация между пользователями или между пользователями и группой эксплуатации?
- Задайте вопросы о поддержке интернационализации (i18n) и локализации (L10n), национальных и региональных языках, почтовых адресах, ценах и т. д. Спросите, требуется ли поддержка разных валют.

2. Исходя из категорий пользователей проясните требования к масштабированию. Оцените количество ежедневных активных пользователей, а затем частоту запросов в день или в час. Например, если сервис поиска имеет 1 миллиард ежедневных пользователей, каждый из которых отправляет 10 поисковых запросов, вы получаете 10 миллиардов ежедневных запросов, или 420 миллионов ежечасных запросов.
3. Какие данные будут доступны для тех или иных пользователей? Обсудите роли и механизмы аутентификации и авторизации. Обсудите содержимое тела ответа конечной точки API. Затем обсудите, как будет происходить получение данных — в реальном времени, в виде ежемесячных отчетов или с другой частотой?
4. Поиск. Каковы возможные сценарии использования, в которых задействован поиск?
5. Аналитика — одно из типичных требований. Обсудите возможные требования к машинному обучению, включая поддержку экспериментов (например, A/B-тестирования или многоруких бандитов). Введение в эти темы см. по адресам https://www.optimizely.com/optimization-glossary/ab-testing/ и https://www.optimizely.com/optimization-glossary/multi-armed-bandit/.

6. Запишите сигнатуры функций псевдокода (например, `fetchPosts(userId)`) для получения постов по пользователям и сопоставьте их с историями пользователей. Обсудите с экспертом, какие требования необходимы, а какие нет.

Всегда спрашивайте: «Есть ли другие пользовательские требования?» и обсуждайте их. У эксперта не должно возникнуть впечатления, что вы хотите, чтобы он думал за вас или сообщил вам все требования.

Требования не очевидны, и соискатель часто упускает подробности, даже если он думает, что все уточнил. Одна из причин применения agile-практик в разработке заключается в том, что требования трудно или невозможно сформулировать. Новые требования или ограничения в ходе разработки обнаруживаются постоянно. По мере получения опыта соискатель понимает, какие уточняющие вопросы необходимо задать.

Покажите, что вы понимаете, что система может быть расширена для удовлетворения других функциональных требований в будущем, и обсудите их.

Эксперт не ожидает от вас исчерпывающих знаний предметной области, так что вы можете пропустить некоторые требования, для которых необходимы конкретные знания. Вы должны продемонстрировать критическое мышление, внимание к деталям, сдержанность и готовность к обучению.

Затем обсудите нефункциональные требования. Подробно о них говорится в главе 3. Возможно, система должна обслуживать весь мир, поскольку продукт доминирует на глобальном рынке. Выясните у эксперта, нужно ли проектировать с прямым расчетом на масштабирование. Если нет, его может больше интересовать, как вы рассматриваете сложные функциональные требования. Сюда относятся проектируемые модели данных. После обсуждения требований можно переходить к обсуждению проектирования системы.

2.2. ПРЕДВАРИТЕЛЬНАЯ СПЕЦИФИКАЦИЯ API

На основании функциональных требований определите данные, которые пользователи системы ожидают получать и отправлять системе. Обычно у соискателя уходит менее 5 минут на составление эскиза конечных точек GET, POST, PUT и DELETE, включая параметры запроса и пути. В общем случае не рекомендуется подолгу задерживаться на проектировании конечных точек. Скажите эксперту, что за 50 минут вам нужно еще очень многое обсудить, и предложите долго не останавливаться на этой теме.

Вы должны уточнить функциональные требования до планирования конечных точек. Часть собеседования, посвященная этому, уже пройдена, и заниматься этим на данной стадии не стоит, если только вы не упустили что-то важное.

Затем предложите спецификацию API и опишите, насколько она удовлетворяет функциональным требованиям, после чего кратко обсудите ее и определите все функциональные требования, которые вы могли упустить.

2.2.1. Типичные конечные точки API

На этом этапе определяются типичные конечные точки большинства систем. Вы можете быстро пройтись по этим конечным точкам и объяснить, что вы не будете на них останавливаться. До их подробного обсуждения дело вряд ли дойдет, но всегда полезно показать, что вы уделяете внимание деталям, не упуская при этом из виду общую картину.

Контроль исправности

GET /health — тестовая конечная точка. Ответ 4xx или 5xx означает, что в системе возникли проблемы на продакшен. Точка может выдавать простой запрос к базе данных или возвращать такую информацию, как свободное дисковое пространство, статусы других конечных точек и результаты проверки логики приложения.

Регистрация и вход (аутентификация)

Пользователь приложения обычно должен зарегистрироваться (POST /signup) и выполнить вход (POST /login), прежде чем отправлять информацию приложению. OpenID Connect — типичный протокол аутентификации, рассматриваемый в приложении Б.

Управление пользователями и контентом

Также могут понадобиться конечные точки для получения, изменения и удаления информации о пользователях. Многие потребительские приложения предоставляют каналы, по которым пользователи могут помечать/сообщать о неподходящем контенте, например таком, который нарушает закон или правила сообщества.

2.3. СВЯЗИ И ПРОЦЕССЫ ПОЛЬЗОВАТЕЛЕЙ И ДАННЫХ

В разделе 2.1 были упомянуты типы пользователей и данных, а также то, какие данные должны быть доступны тем или иным пользователям. В разделе 2.2 мы спроектировали конечные точки API для выполнения операций CRUD (Create, Read, Update и Delete, то есть создание, чтение, обновление и удаление). Теперь можно нарисовать диаграммы для представления связей между пользователями и данными, а также компонентов системы и процессов обработки данных, происходящих в них.

Этап 1

- Нарисуйте прямоугольник для каждого типа пользователя.
- Нарисуйте прямоугольник для каждой системы, обеспечивающей выполнение функциональных требований.
- Нарисуйте связи между пользователями и системами.

Этап 2

- Разделите обработку запросов и хранение данных.
- Создайте разные дизайны, основанные на нефункциональных требованиях (например, обработка в реальном времени и согласованность в конечном счете).
- Добавьте общие сервисы.

Этап 3

- Разбейте систему на компоненты — обычно библиотеки или сервисы.
- Нарисуйте связи.
- Добавьте функциональность ведения журналов, мониторинга и оповещения.
- Добавьте средства безопасности.

Этап 4

- Добавьте сводное описание дизайна системы.
- Укажите все необходимые дополнительные требования.
- Проанализируйте отказоустойчивость системы. Что может пойти не так с каждым компонентом? Сетевые задержки, рассогласования, отсутствие линеаризуемости. Что можно сделать для предотвращения и/или преодоления каждой ситуации и повышения отказоустойчивости этого компонента и системы в целом?

В приложении B приведен обзор *модели C4* — метода построения диаграмм системной архитектуры, основанного на декомпозиции системы по разным уровням абстракции.

2.4. ПРОЕКТИРОВАНИЕ МОДЕЛИ ДАННЫХ

Уточните, будете ли вы проектировать модель данных с нуля или же воспользуетесь существующими базами данных. Совместное использование баз данных сервисами обычно считается антипаттерном, так что если вы обращаетесь к существующим базам данных, стоит построить больше конечных точек API, спроектированных для программных клиентов, а также пакетные и/или потоковые пайплайны ETL, направленные от БД/к БД.

Ниже перечислены типичные проблемы, которые могут возникать с общими базами данных:

- Запросы от разных сервисов к одним и тем же таблицам могут конкурировать за ресурсы. Некоторые запросы (например, UPDATE для множества записей, или транзакции, содержащие другие продолжительные запросы) могут блокировать таблицу на длительное время.
- Усложнение миграции схемы. Миграция схемы, которая совершается в интересах одного сервиса, может нарушить код интерфейса DAO (data access object) других сервисов. Это означает, что даже если инженер работает только с этим сервисом, ему придется обращать внимание на обновление низкоуровневых подробностей бизнес-логики и, возможно, даже исходного кода других сервисов, с которыми он не работает. Это приведет к нерациональному использованию как своего времени, так и времени других инженеров, которые вносят эти изменения и должны сообщать о них остальным. Больше времени будет потрачено на составление и чтение документации, создание слайдов презентаций и совещания. Согласование в командах предлагаемых схем миграций может стать напрасной тратой времени инженеров, поскольку есть вероятность, что другие команды не согласуют предложенные схемы миграции или не добьются компромисса по некоторым изменениям, а это приведет к возникновению технического долга и снижению общей продуктивности.
- Разные сервисы, совместно использующие один набор баз данных, ограничены использованием конкретных технологий (например, MySQL, HDFS, Cassandra, Kafka и т. д.) независимо от того, насколько хорошо эти технологии подходят для сценариев использования каждого сервиса. Сервисы не могут выбрать технологию, которая лучше подходит для их требований.

Это означает, что в каждом случае для сервиса приходится проектировать новую схему. Можно использовать тела запроса и ответа конечных точек API, рассмотренные в предыдущем разделе, в качестве отправных точек для проектирования схемы; при этом каждое тело тесно привязывается к схеме таблицы — возможно, с объединением тел запросов чтения (GET) и записи (POST и PUT) одних путей к одной таблице.

2.4.1. Пример. Недостатки совместного использования баз данных несколькими сервисами

Представьте, что вы проектируете систему электронной коммерции. Вам может понадобиться сервис для получения данных бизнес-метрик, например общего количества заказов за последние семь дней. Ваши команды обнаружили, что без источника достоверных данных для определения бизнес-метрик разные команды вычисляли метрики по-разному. Например, должно ли общее количество заказов включать отмененные заказы или заказы, по которым был произведен возврат средств? Какой часовой пояс использовать для определения порога

«семь дней назад»? Включают ли «последние семь дней» сегодняшний день? Коммуникации между командами для уточнения метрик оказываются слишком затратными и ненадежными.

И хотя вычисление бизнес-метрик использует данные заказов из сервиса Orders, вы решаете сформировать новую команду для создания специализированного сервиса Metrics, так как определения метрик могут изменяться независимо от данных заказов.

Сервис Metrics будет зависеть от сервиса Orders, к которому он обращается за данными заказов. Запрос метрики будет обрабатываться следующим образом:

1. Получение метрики.
2. Получение соответствующих данных от сервиса Orders.
3. Вычисление метрики.
4. Возвращение значения метрики.

Если оба сервиса совместно используют одну базу данных, то при вычислении метрик создаются запросы SQL к таблицам сервиса Orders. Это усложняет миграцию схемы. Допустим, команда Orders решила, что пользователи таблицы Order отправляют к ней слишком много больших запросов. После краткого анализа команда решает, что запросы по недавним заказам более важны и задержка для них должна быть меньше, чем для запросов по старым заказам. Команда предлагает хранить в таблице Order заказы только за последний год, а более старые заказы переместить в таблицу Archive. Таблице Order может быть выделено большее количество чтений на ближайшей реплике (follower reads), чем таблице Archive.

Команда Metrics должна понять суть предлагаемых изменений и изменить вычисление метрик, чтобы оно учитывало данные обеих таблиц. Возможно, команда Metrics будет возражать против предлагаемых изменений, так что предложение может быть отклонено, и добиться прироста производительности за счет ускорения запросов к данным недавних заказов не удастся.

Если команда Orders захочет переместить таблицу Order в Cassandra для снижения задержки записи, а сервис Metrics продолжит работать на SQL из-за его простоты и более низкой частоты записи, сервисы не смогут совместно использовать одну базу данных.

2.4.2. Как предотвратить конфликты одновременных обновлений

Возможны ситуации, в которых клиентское приложение разрешает некоторым пользователям редактировать общую конфигурацию. Если операция редактирования представляет сложность для пользователя (он тратит больше нескольких

секунд на ввод информации до отправки изменений), может возникнуть риск, что несколько пользователей одновременно отредактируют конфигурацию, а затем перезапишут чужие изменения при сохранении. Системы контроля версий предотвращают эту опасность для исходного кода, но многие сценарии предполагают участие нетехнических пользователей; конечно, нельзя рассчитывать на то, что они знают git.

Например, сервис бронирования номеров в отелях требует ввода дат заселения и выезда, а также контактных и платежных данных и только после этого отправляет запросы на бронирование. Необходимо позаботиться о том, чтобы номер не был забронирован сразу несколькими пользователями.

Другой пример — настройка содержимого пуш-уведомлений. Например, мы можем предоставить сотрудникам браузерное приложение, в котором они будут настраивать активные уведомления, отправляемые нашему приложению Beigel (см. главу 1). Конкретная конфигурация пуш-уведомлений может принадлежать какой-то команде. Необходимо позаботиться о том, чтобы разные члены команды не редактировали уведомления одновременно и не перезаписывали изменения других участников.

Существуют разные способы избежать одновременных обновлений. Мы рассмотрим один из них в этом разделе.

Для предотвращения таких ситуаций можно заблокировать конфигурацию во время редактирования. Сервис может содержать таблицу SQL для хранения этих конфигураций. Можно включить в соответствующую таблицу SQL столбец с меткой времени, который будет называться `unix_locked`, и строковые столбцы `edit_username` и `edit_email`. (Такой дизайн схемы не нормализован, но на практике обычно допускается. Спросите эксперта, настаивает ли он на использовании нормализованной схемы.) Затем можно предоставить конечную точку PUT, которую пользовательский интерфейс может использовать для оповещения бэкенда, когда пользователь щелкает на значке или кнопке редактирования, чтобы начать редактирование строки запроса. На рис. 2.1 изображена последовательность действий в ситуации, когда два пользователя решают отредактировать конфигурацию пуш-уведомлений примерно в одно и то же время. Один пользователь может заблокировать конфигурацию на определенный период (например, 10 минут), а другой пользователь обнаруживает, что она заблокирована:

1. Алиса и Боб просматривают конфигурацию пуш-уведомлений в браузерном приложении Notifications. Алиса решает заменить заголовок Celebrate National Bagel Day![1] на 20% off on National Bagel Day![2] Она щелкает на кнопке Edit (Редактировать). При этом происходит следующее:

[1] Отмечаем День рогалика! (*англ.*) — *Примеч. ред.*

[2] Получите скидку 20 % в День рогалика! (*англ.*) — *Примеч. ред.*

a) Событие щелчка отправляет запрос PUT, который посылает имя пользователя и адрес электронной почты на бэкенд. Балансировщик нагрузки бэкенда назначает этот запрос хосту.

b) Хост бэкенда Алисы выдает два запроса SQL, по одному за раз. Сначала он определяет текущее время `unix_locked`:

```
SELECT unix_locked FROM table_name WHERE config_id = {config_id}.
```

c) Бэкенд проверяет, что метка времени `edit_start` находится не более чем на 12 минут в прошлом (2-минутный буфер добавлен на случай, если таймер обратного отсчета на шаге 2 запустился поздно и если синхронизация часов хостов не идеальна). Если условие соблюдено, строка обновляется обозначением блокировки конфигурации. Запрос UPDATE сохраняет в `edit_start` текущее время UNIX бэкенда и заменяет содержимое `edit_username` и `edit_email` именем пользователя и адресом электронной почты Алисы. Фильтр `unix_locked` нужен просто на случай, если другой пользователь меняет его в это время. Запрос UPDATE возвращает логический признак, указывающий, что он выполнен успешно:

```
UPDATE table_name SET unix_locked = {new_time}, edit_username = {username},
edit_email = {email} WHERE config_id = {config_id} AND unix_locked =
{unix_locked}
```

d) Если запрос UPDATE выполнен успешно, бэкенд передает пользовательскому интерфейсу код успеха 200 с телом ответа вида `{"can_edit": "true"}`.

2. Пользовательский интерфейс открывает страницу, в которой Алиса может вносить правку, и запускает 10-минутный таймер отсчета. Она стирает старый заголовок и начинает вводить новый.

3. Между запросами SQL на шагах 1.2 и 1.3 Боб тоже решает отредактировать конфигурацию:

a) Он щелкает на Edit, выдавая запрос PUT, который назначается другому хосту.

b) Первый запрос SQL возвращает то же время `unix_locked`, что и на шаге 1.2.

c) Второй запрос SQL отправляется сразу же после запроса на шаге 1.3. Запросы SQL DML отправляются одному хосту (см. раздел 4.3.2). Это означает, что запрос не должен выполняться до завершения запроса с шага 1.3. При выполнении запроса значение `unix_time` изменилось, поэтому строка данных не обновляется и сервис SQL возвращает на бэкенд ответ `false`. Бэкенд возвращает пользовательскому интерфейсу код успеха 200 с телом ответа вида `{"can_edit": "false", "edit_start": "1655836315", "edit_username": "Alice", "edit_email": "alice@beigel.com"}`.

d) Пользовательский интерфейс вычисляет количество минут, оставшихся у Алисы, и выводит уведомление с текстом вида «Алиса (alice@beigel.com) вносит изменения. Попробуйте снова через 8 минут».

4. Алиса завершает правку и щелкает на кнопке Save (Сохранить). Кнопка отправляет запрос PUT на бэкенд, который сохраняет внесенные изменения и стирает данные `unix_locked`, `edit_start`, `edit_username` и `edit_email`.
5. Боб снова щелкает на Edit и переходит к внесению изменений. Если Боб щелкнул на Edit по крайней мере через 12 минут после значения `edit_start`, он также сможет вносить изменения. Если Алиса не сохранила свои изменения до истечения обратного отсчета, то пользовательский интерфейс выдаст уведомление о том, что она больше не сможет их сохранить.

А что, если Боб зайдет на страницу конфигурации пуш-уведомлений после того, как Алиса начнет редактировать конфигурацию? Одна из возможных

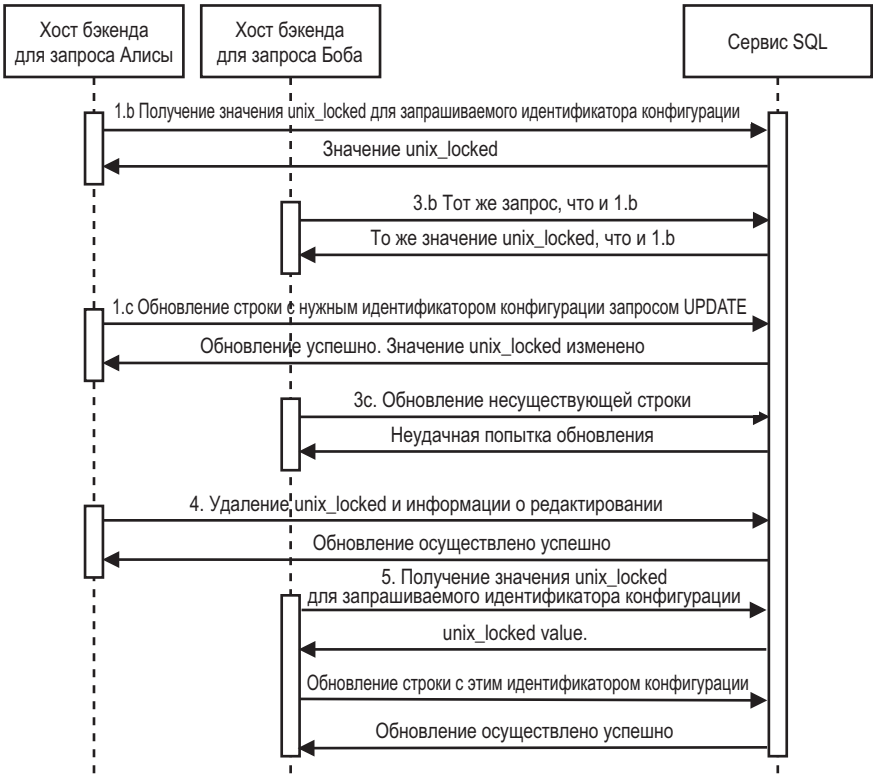

Рис. 2.1. Механизм блокировки, использующий SQL. Два пользователя обращаются с запросами на обновление одной строки SQL, соответствующей одному идентификатору конфигурации. Хост Алисы сначала получает значение метки времени unix_locked, после чего отправляет запрос UPDATE для обновления этой строки; тем самым Алиса блокирует конкретный идентификатор конфигурации. Сразу после того, как ее хост отправил этот запрос на шаге 1.3, хост Боба тоже отправил запрос UPDATE, но хост Алисы изменил значение unix_locked, поэтому запрос UPDATE Боба не может быть выполнен и Боб не сможет заблокировать этот идентификатор конфигурации

оптимизаций пользовательского интерфейса в этом случае — блокировка кнопки Edit и вывод уведомления, чтобы Боб знал, что он не может редактировать конфигурацию, потому что сейчас ее редактирует Алиса. Чтобы реализовать этот вариант, можно добавить три поля в ответ GET для конфигурации пуш-уведомлений; пользовательский интерфейс обработает эти поля и отрендерит кнопку Edit в «активном» или «заблокированном» состоянии.

Узнать больше о контроле версий с использованием Jakarta Persistence API и Hibernate можно по адресу https://vladmihalcea.com/jpa-entity-version-property-hibernate/.

2.5. ВЕДЕНИЕ ЖУРНАЛОВ, МОНИТОРИНГ И ОПОВЕЩЕНИЯ

Темам ведения журналов (логирования), мониторинга и оповещений посвящено много книг. В этом разделе мы рассмотрим ключевые концепции, которые необходимо упомянуть в интервью, а также остановимся на тех из них, которые, скорее всего, будут обсуждаться подробно. Обязательно затроньте на собеседовании тему мониторинга.

2.5.1. Важность мониторинга

Мониторинг абсолютно необходим в любой системе, поскольку он дает представление об опыте взаимодействия с пользователем. Данные мониторинга помогут выявить ошибки, снижение производительности, непредвиденные события и другие слабые места системы, препятствующие удовлетворению текущих и возможных будущих функциональных и нефункциональных требований.

Отказ веб-сервисов может произойти в любой момент. Эти отказы можно классифицировать по степени важности и по тому, насколько срочно они требуют внимания. Критичным сбоям необходимо уделять внимание немедленно. Некритичные сбои могут подождать, пока не будут завершены задачи с более высоким приоритетом. Уровни срочности определяются требованиями и здравым смыслом. Если сервис является зависимостью для других сервисов, то он может служить потенциальным источником снижения их производительности; следовательно, необходимо создать конфигурацию ведения журналов и мониторинга, которая позволит выявлять возможное снижение производительности и отвечать на возникающие вопросы.

2.5.2. Наблюдаемость

Это приводит нас к концепции наблюдаемости. Наблюдаемость системы является метрикой того, насколько хорошо она оснащена и насколько легко узнать, что в ней происходит (John Arundel & Justin Domingus, Cloud Native DevOps

with Kubernetes, p. 272, O'Reilly Media Inc, 2019). Без ведения журналов, метрик и трассировки система не прозрачна. Не получится легко определить, насколько хорошо изменение кода, предназначенное для снижения 99-го процентиля конкретной конечной точки на 10 %, работает на продакшен. Если 99-й процентиль снижается намного меньше, чем на 10 %, или намного больше, чем на 10%, необходимо на основании инструментальных данных суметь сделать выводы о том, почему прогнозы не оправдались.

Подробное обсуждение четырех главных сигналов мониторинга: задержки, трафика, ошибок и насыщения — содержится в руководстве Google по SRE[1] (https://sre.google/sre-book/monitoring-distributed-systems/#xref_monitoring_golden-signals).

1. *Задержка* — можно настроить оповещения для задержки, превышающей значения из SLA, — например, более 1 секунды. SLA может определять условия для любого отдельного запроса продолжительностью более 1 секунды, или же оповещение может срабатывать для 99-го процентиля по скользящему окну (например, 5 секунд, 10 секунд, 1 минута, 5 минут).
2. *Трафик* — измеряется в запросах HTTP в секунду. Можно настроить для разных конечных точек оповещения, срабатывающие при слишком высоком объеме трафика. Конкретные числовые значения устанавливаются на основании предельной нагрузки, определяемой в процессе нагрузочного тестирования.
3. *Ошибки* — следует установить срочные оповещения для кодов ответов 4xx или 5xx, которыми необходимо заняться немедленно. Обеспечить выдачу некритичных оповещений (или критичных, в зависимости от требований) при ошибках аудита.
4. *Насыщение* — в зависимости от того, ограничено ли процессорное время, память или ввод/вывод системы, можно установить целевые значения загруженности, которые не должны превышаться. Можно установить оповещения, срабатывающие при достижении целевых значений загруженности. Другой пример — загруженность хранилища данных. Можно установить оповещение, которое отправляется, если до исчерпания объема хранилища (из-за использования файлов или базы данных) осталось несколько часов или дней.

Три инструмента мониторинга и оповещений: метрики, дашборды и сигналы. *Метрика* — переменная, значение которой можно измерить (например, количество ошибок, задержка или время обработки). *Дашборд* выводит сводное представление основных метрик сервиса. *Сигнал* представляет собой оповещение, отправляемое владельцам сервиса в ответ на некоторую проблему, возникшую с сервисом. Метрики, дашборды и сигналы заполняются на основании обработки данных журналов. Для создания этих инструментов и управления ими можно использовать стандартный браузерный UI.

---

[1] Site Reliability Engineering, обеспечение надежности информационных систем. — *Примеч. пер.*

Такие метрики ОС, как загруженность процессора, степень использования памяти, степень использования диска и сетевой ввод/вывод, могут размещаться на дашборде и использоваться для того, чтобы оптимизировать распределение оборудования для сервиса или выявлять утечки памяти.

Фреймворк бэкенда может выполнять регистрацию каждого запроса по умолчанию или предоставлять простые аннотации для методов запросов, чтобы включить логирование. Вы можете добавлять команды логирования в код приложения. Также можно вручную регистрировать в коде значения некоторых переменных, которые помогут понять, как обрабатывается запрос клиента.

В книге Шолля и др. (Boris Scholl, Trent Swanson & Peter Jausovec. Cloud Native: Using Containers, Functions, and Data to build Next-Generation Applications. O'Reilly, 2019, p. 145) перечислены следующие общие соображения по организации ведения журналов:

- Логи должны иметь структуру, упрощающую парсинг с использованием инструментов и автоматизированных средств.
- Каждый лог должен иметь уникальный идентификатор для отслеживания запросов между сервисами, а также совместного использования пользователями и разработчиками.
- Логи должны быть краткими, удобочитаемыми и содержательными.
- Метки времени должны использовать одинаковый формат часового пояса и времени. Журнал, содержащий записи с разными форматами часовых поясов и времени, трудно читать и парсить.
- Разбейте логи на категории. Начните с категорий отладочных сообщений, информационных сообщений и ошибок.
- Не сохраняйте в журнале конфиденциальную информацию — пароли, строки подключения и т. д. Такая информация часто обозначается сокращением PII (Personally Identiiable Information, то есть «персональные данные»).

Следующие журналы типичны для большинства сервисов. Многие средства ведения журналов на уровне запросов сохраняют эту информацию в конфигурациях по умолчанию:

- Ведение журналов работы хостов:
  - Загруженность процессора и памяти на хостах.
  - Сетевой ввод/вывод.
- Журналы уровня запросов сохраняют подробную информацию о каждом запросе:
  - Задержка.
  - Кто и когда выдал запрос.
  - Имя функции и номер строки.

- Путь и параметры запроса, заголовки и тело.
- Возвращаемый код статуса и тело (с возможными сообщениями об ошибках).

В конкретной системе некоторые параметры пользовательских взаимодействий могут быть особенно важны, например, ошибки. Можно включить в приложение команды записи в журнал и настроить специализированные метрики, дашборды и сигналы, отслеживающие эти взаимодействия. Например, если для вас важно отслеживать ошибки 5xx, вызванные сбоями приложений, можно создать метрики, дашборды и сигналы, обрабатывающие некоторые подробности, такие как параметры запросов и возвращаемые коды статуса и сообщения об ошибках, если они есть.

Также следует сохранять в журнале события для мониторинга того, насколько хорошо система удовлетворяет уникальным функциональным и нефункциональным требованиям. Например, при создании кэша в журнале можно сохранять данные о количестве отказов кэша, промахов и попаданий в кэш. Соответственно, метрики должны включать количество отказов, промахов и попаданий.

В корпоративных системах пользователям иногда предоставляется доступ к данным мониторинга — вплоть до создания средств мониторинга, предназначенных специально для пользователей. Например, клиенты могут создавать дашборды для отслеживания состояний своих запросов, фильтрации и агрегирования метрик и сигналов по категориям (например, путям URL).

Также следует обсудить порядок обработки сбоев без видимых проявлений. Они могут быть вызваны как ошибками в коде приложения, так и зависимостями (например, библиотеками и другими сервисами, которые допускают код ответа 2xx, когда он должен быть равен 4xx или 5xx) или просто указывать на то, что сервис нуждается в улучшении средств ведения журналов и мониторинга.

Кроме ведения журналов, мониторинга и оповещений для отдельных запросов, можно создать пакетные и потоковые задания аудита для проверки данных системы. Это напоминает мониторинг целостности данных системы. Можно создать оповещения, которые срабатывают, если результаты задания указывают на сбой при проверке. Такая система рассматривается в главе 10.

2.5.3. Реакция на оповещения

Команда, занимающаяся разработкой и обслуживанием сервиса, обычно состоит из нескольких инженеров. Эта команда может определить график обслуживания по требованию для срочных оповещений сервиса. Инженер может не разбираться во всех тонкостях причины конкретного оповещения, поэтому придется подготовить ранбук (пошаговую инструкцию по выполнению типовых задач) со списком оповещений, возможных причин и процедур для поиска и исправления проблем.

Если при подготовке ранбука окажется, что некоторые инструкции состоят из серии команд, которые для решения проблемы легко скопировать/вставить (например, перезапуска хоста), эти шаги, как и ведение журналов для них, должны быть автоматизированы в приложении (Mike Julian, Practical Monitoring, глава 3, O'Reilly Media Inc, 2017). Ранбук по возможности должен содержать описание автоматизированного восстановления после сбоев. Если некоторые инструкции состоят из выполнения команд для просмотра конкретных метрик, эти метрики должны выводиться на дашборд.

В компании может работать SRE-команда, состоящая из инженеров, которые разрабатывают инструменты и процессы для обеспечения высокой надежности критических сервисов. Если сервис получает покрытие SRE, возможно, его сборка должна пройти проверку на соответствие критериям команды SRE перед развертыванием. Критерии обычно включают высокую степень покрытия модульными тестами, комплекс функциональных тестов, проходящих обзор SRE, и хорошо написанную документацию с полным покрытием и описанием возможных проблем, проверенную командой SRE.

После того как сбой будет устранен, желательно составить заключение, в котором указать, что пошло не так и что было сделано, чтобы гарантировать, что сбой не повторится. Заключение не должно быть направлено на поиск виновных, иначе сотрудники будут стараться скрывать проблемы вместо того, чтобы решать их.

На основании закономерностей, выявленных в действиях, которые предпринимаются для устранения проблем, можно определить способы автоматизации устранения таких проблем, включая самовосстановление системы.

2.5.4. Инструменты ведения журналов на уровне приложения

Комплекс ELK с открытым исходным кодом (Elasticsearch, Logstash, Beats, Kibana) и платный сервис Splunk — стандартные средства ведения журналов на уровне приложения. Logstash используется для сбора логов и управления ими. Elasticsearch — поисковое ядро, используемое для хранения, индексирования и поиска по логам. Kibana используется для визуализации и вывода логов на дашбордах с Elasticsearch в качестве источника данных. Beats добавился в 2015 году как облегченный поставщик данных для Elasticsearch или Logstash в реальном времени.

В этой книге везде, где мы говорим о ведении журналов любого события, имеется в виду, что событие регистрируется стандартным сервисом ELK, который используется для ведения журналов и другими сервисами в организации.

Существуют разные средства мониторинга — как проприетарные, так и распространяемые на условиях FOSS (Free and Open-Source Software, свободное ПО с открытым исходным кодом). Мы кратко рассмотрим некоторые из этих

инструментов, но их полный список, подробное описание и сравнение выходят за рамки данной книги.

Эти инструменты различаются по своим характеристикам:

- Функциональность. Различные инструменты могут предоставлять полный набор или подмножество средств ведения журналов, мониторинга, оповещения и отображения дашбордов.
- Поддержка разных операционных систем и других типов оборудования, кроме серверов: балансировщиков нагрузки, коммутаторов, модемов, маршрутизаторов, сетевых адаптеров и т. д.
- Потребление ресурсов.
- Популярность, которая прямо пропорциональна простоте поиска инженеров, знакомых с системой.
- Поддержка разработчиков, например частота обновлений.

Также они различаются по субъективным характеристикам:

- Сложность освоения.
- Сложность ручной настройки и вероятность ошибок у новых пользователей.
- Простота интеграции с другими программными продуктами и сервисами.
- Количество и критичность ошибок.
- UX. У некоторых инструментов имеются браузерные или десктопные клиенты, и пользователи могут предпочитать один UI другому.

Некоторые средства мониторинга FOSS:

- *Prometheus* + *Grafana* — Prometheus для мониторинга, Grafana для визуализации и представления дашбордов.
- *Sensu* — система мониторинга, использующая Redis для хранения данных. Можно настроить Sensu для отправки сигналов стороннему сервису оповещений.
- *Nagios* — система мониторинга и оповещений.
- *Zabbix* — система мониторинга, включающая средства отображения дашбордов.

Среди проприетарных инструментов стоит выделить Splunk, Datadog и New Relic.

База данных временны́х рядов (TSDB, Time Series Database) — система, оптимизированная для хранения и предоставления временных рядов — например, непрерывной записи с ведением журналов данных временных рядов. Например, многие запросы выполняются для последних данных, так что старые данные утрачивают ценность, и для экономии памяти можно настроить прореживание

данных в TSDB. При этом старые данные обобщаются посредством вычисления средних значений за определенный интервал. Сохраняются только эти средние значения, а исходные данные удаляются, что позволяет сэкономить память. Период хранения данных и детализация зависят от требований и бюджета.

Для дальнейшего снижения затрат на хранение старых данных можно сжать их или воспользоваться дешевыми носителями, например магнитными лентами или оптическими дисками. Примеры нестандартных конфигураций (например, серверов на жестких дисках, которые замедляются или останавливаются во время бездействия) представлены в материалах https://www.zdnet.com/article/could-the-tech-beneath-amazons-glacier-revolutionise-data-storage/ или https://arstechnica.com/information-technology/2015/11/to-go-green-facebook-puts-petabytes-of-cat-pics-on-ice-and-likes-windfarming/:

- *Graphite* — часто используется для записи метрик ОС (хотя также может использоваться для мониторинга других конфигураций, например веб-сайтов и приложений), для наглядного представления которых применяется веб-приложение Grafana.
- *Prometheus* — также обычно используется с Grafana для визуализации.
- *OpenTSDB* — распределенная масштабируемая база данных TSDB, использующая HBase.
- *InluxDB* — TSDB с открытым кодом, написанная на языке Go.

Prometheus — система мониторинга с открытым исходным кодом, построенная на основе TSDB. Prometheus запрашивает метрики из конечных точек HTTP, а Pushgateway отправляет сигналы в Alertmanager, где можно настроить отправку по различным каналам (например, электронной почте или PagerDuty). Язык запросов Prometheus (PromQL) используется для анализа метрик и построения графиков.

Nagios — проприетарный инструмент мониторинга IT-инфраструктуры, предназначенный для мониторинга серверов, сетей и приложений. Для него существуют сотни сторонних плагинов, веб-интерфейсов и расширенных средств визуализации и отображения дашбордов.

2.5.5. Потоковый и пакетный аудит качества данных

Качество данных — неформальный термин, который означает, что данные относятся к реальному объекту и могут использоваться для намеченных целей. Например, если в конкретной таблице, обновляемой заданием ETL, отсутствуют некоторые строки, которые были этим заданием сгенерированы, качество данных будет низким.

Таблицы баз данных могут проходить непрерывный и/или периодический аудит для обнаружения проблем с качеством данных. Такой аудит можно реализовать

посредством потоковых и пакетных заданий ETL для проверки недавно добавленных и измененных данных. Это особенно полезно для обнаружения ошибок, которые остались незамеченными в ходе предыдущих проверок, например проведенных во время обработки запроса к сервису.

Концепцию можно расширить до гипотетического общего сервиса пакетного аудита баз данных (эта тема рассматривается в главе 10).

2.5.6. Обнаружение аномалий данных

Обнаружение аномалий — концепция машинного обучения (МО), предназначенная для обнаружения нетипичных точек данных. Полное описание концепций МО в этой книге мы приводить не будем. В этом разделе кратко описано обнаружение аномалий для контроля качества данных и генерации аналитических выводов (например, неожиданный рост или снижение определенной метрики может указывать на проблемы с обработкой данных или изменение рыночных условий).

В простейшей форме обнаружение аномалий представляет собой подачу непрерывного потока данных на вход алгоритма обнаружения аномалий. После обработки определенного количества точек данных, которое в области МО называется обучающим набором, алгоритм обнаружения аномалий строит статистическую модель. Задача этой модели — получить точку данных и вернуть вероятность того, что эта точка данных является аномальной. Работоспособность модели можно проверить на специальном наборе точек данных, каждая точка в котором вручную помечена как нормальная или аномальная. Наконец, можно оценить характеристики модели в числовом виде, протестировав ее на другом вручную размеченном тестовом наборе.

Многие параметры могут настраиваться вручную, например используемые модели МО, количество точек данных в каждом из трех наборов и параметры модели для регулировки характеристик, например точности и полноты. Такие концепции МО, как точность и полнота, выходят за рамки этой книги.

На практике такой подход к обнаружению аномалий данных оказывается сложным и затратным в реализации, обслуживании и использовании. Его применение следует ограничить критическими наборами данных.

2.5.7. Скрытые ошибки и аудит

Скрытые ошибки могут происходить из-за багов программной логики, когда конечная точка возвращает код статуса 200 даже при возникновении ошибки. Можно написать пакетные задания ETL для аудита недавних изменений в базах данных и выдавать оповещения при неудачах аудита. Подробности см. в главе 10.

2.5.8. Дополнительная литература по теме наблюдаемости

- Michael Hausenblas. Cloud Observability in Action, Manning Publications, 2023. Руководство по применению практик наблюдаемости с облачными бессерверными средами и средами на базе Kubernetes.
- https://www.manning.com/liveproject/conigure-observability. Практический курс реализации средств шаблонов сервиса, связанных с наблюдаемостью.
- Mike Julian, Practical Monitoring, O'Reilly Media Inc, 2017. Книга, посвященная лучшим практикам наблюдаемости, реагированию на инциденты и антипаттернам.
- Boris Scholl, Trent Swanson, and Peter Jausovec. Cloud Native: Using Containers, Functions, and Data to build Next-Generation Applications. O'Reilly, 2019. В книге сделан упор на то, что наблюдаемость — неотъемлемое свойство облачно-ориентированных приложений.
- John Arundel and Justin Domingus. Cloud Native DevOps with Kubernetes[1], главы 15 и 16, O'Reilly Media Inc, 2019. В этих главах рассматриваются темы наблюдаемости, мониторинга и метрик в облачно-ориентированных приложениях.

2.6. ПАНЕЛЬ ПОИСКА

Поиск стал стандартной функцией многих приложений. Многие интерфейсные приложения предоставляют панели поиска, чтобы пользователь мог быстро найти нужные данные. Данные могут индексироваться в кластерах Elasticsearch.

2.6.1. Введение

Панель поиска является стандартным UI-компонентом многих приложений. Она может состоять из единственной строки поиска или содержать другие средства фильтрации. На рис. 2.2 приведен пример панели поиска.

Рис. 2.2. Панель поиска Google с раскрывающимися меню для фильтрации результатов (изображение предоставлено Google)

[1] Арундел Д., Домингус Д. «Kubernetes для DevOps: развертывание, запуск и масштабирование в облаке». СПб., издательство «Питер».

Стандартные методы реализации поиска:

1. Поиск по базе данных SQL с оператором `LIKE` и шаблонами. Запрос может выглядеть примерно так: `SELECT <column> FROM <table> WHERE Lower(<column>) LIKE "%Lower(<search_term>)%"`.
2. Использование специализированных библиотек, таких как match-sorter (https://github.com/kentcdodds/match-sorter), — библиотека JavaScript, которая получает поисковый запрос и выполняет поиск и сортировку записей. Такое решение реализуется отдельно в каждом клиентском приложении. Это подходящее и технически простое решение для текстовых данных объемом до нескольких гигабайт (то есть до миллионов записей). Веб-приложение обычно загружает свои данные из бэкенда, и эти данные вряд ли будут занимать более нескольких мегабайт, иначе приложение не будет масштабироваться на миллионы пользователей. Мобильное приложение может хранить данные локально, поэтому работа с гигабайтами данных теоретически возможна, но синхронизация данных между миллионами смартфонов может стать нереальной задачей.
3. Использование поискового ядра, такого как Elasticsearch. Это решение хорошо масштабируется и может работать с петабайтами данных.

Первый метод обладает рядом ограничений. Его стоит рассматривать только в качестве быстрой временной реализации, которая вскоре будет удалена или заменена нормальным поисковым ядром. Основные его недостатки:

- Трудность настройки поискового запроса.
- Отсутствие таких сложных функций, как бустинг, веса, нечеткий поиск или предварительная обработка текста (например, выделение корней или разбиение на лексемы).

Это обсуждение исходит из того, что отдельные записи малы по объему, то есть содержат текстовую информацию, а не видео. Для записей с видео индексирование и операции поиска выполняются не с видеоданными, а с сопроводительными текстовыми метаданными. Реализация индексирования и поиска в поисковых ядрах выходит за рамки этой книги.

Описанные методы будут рассмотрены в части 2, где мы уделим намного больше внимания использованию Elasticsearch.

2.6.2. Реализация панели поиска с Elasticsearch

Организация может иметь общий кластер Elasticsearch для обеспечения поисковых потребностей многих своих сервисов. В этом разделе сначала будет описан базовый полнотекстовый поисковый запрос Elasticsearch, а затем основные действия для добавления функциональности Elasticsearch в сервис при

существующем кластере Elasticsearch. Мы не будем подробно рассматривать создание такого кластера или описывать соответствующие концепции и терминологию. В наших примерах будет использоваться приложение Beigel (из главы 1).

Чтобы обеспечить поддержку базового полнотекстового поиска с нечеткой логикой, можно связать панель поиска с конечной точкой GET, которая перенаправляет запрос сервису Elasticsearch. Запрос Elasticsearch направлется к индексу Elasticsearch (по аналогии с базой данных в реляционных БД). Если запрос GET возвращает ответ 2xx со списком результатов поиска, фронтенд загружает страницу результатов, на которой выводится список.

Например, если в приложении Beigel имеется панель поиска и пользователь ищет текст «sesame», запрос Elasticsearch может принимать одну из форм, описанных ниже.

Условие поиска может быть включено в параметр запроса, в этом случае разрешены только точные совпадения:

```
GET /beigel-index/_search?q=sesame
```

Если условие включено в тело запроса JSON, можно использовать все возможности языка предметной области Elasticsearch (данная тема выходит за рамки книги):

```
GET /beigel-index/_search
{
  "query": {z6
    "match": {
      "query": "sesame",
      "fuzziness": "AUTO"
    }
  }
}
```

`"fuzziness": "AUTO"` разрешает применять нечеткий поиск совпадений, который находит много вариантов применения (например, если в условии или результатах поиска присутствуют опечатки).

Результаты возвращаются в виде массива JSON с совпадениями, отсортированными по убыванию соответствия, как в следующем примере. Бэкенд может возвращать результаты на фронтенд, который проведет их парсинг и выведет для пользователя.

2.6.3. Индексирование и поглощение данных в Elasticsearch

Создание индекса Elasticsearch включает поглощение (ingestion) документов, в которых должен проводиться поиск, когда пользователь отправляет запрос из панели поиска, и операцию индексирования. В дальнейшем индекс обновляется

периодическим индексированием, или индексированием по событию, или запросами на удаление через Bulk API.

Чтобы изменить привязку индекса, можно создать новый и удалить старый индекс. Также можно воспользоваться операцией переиндексирования Elasticsearch, но она обойдется дорого, поскольку внутренний коммит Lucene выполняется синхронно после каждого запроса на запись (https://www.elastic.co/guide/en/elasticsearch/reference/current/index-modules-translog.html#index-modules-translog).

Для создания индекса Elasticsearch требуется, чтобы все данные, по которым будет вестись поиск, находились в хранилище документов Elasticsearch, что увеличивает общие требования к пространству хранения данных. Существуют оптимизации, при которых передается только подмножество данных, подлежащих индексированию.

В табл. 2.1 представлено примерное соответствие терминологии SQL и Elasticsearch.

Таблица 2.1. Примерное соответствие терминологии SQL и Elasticsearch. Между соотнесенными терминами существуют различия, и таблицу не следует воспринимать буквально. Она предназначена для новичков Elasticsearch, имеющих опыт работы в SQL, которые могут использовать ее как основу для дальнейшего изучения.

SQL	Elasticsearch
База данных	Индекс
Раздел	Сегмент
Таблица	Тип (считается устаревшим, замены нет)
Столбец	Поле
Строка	Документ
Схема	Отображение (маппинг)
Индекс	Индексируются все данные

2.6.4. Использование Elasticsearch вместо SQL

Elasticsearch может использоваться как SQL. В Elasticsearch существует концепция контекста запроса и контекста фильтра (https://www.elastic.co/guide/en/elasticsearch/reference/current/query-ilter-context.html). Согласно документации, в контексте фильтра предложение запроса отвечает на вопрос: «Соответствует ли этот документ этому предложению запроса?» Ответ представляет собой простое «да» или «нет»; никакие метрики не вычисляются. В контексте запроса предложение запроса отвечает на вопрос: «Насколько хорошо этот документ соответствует этому предложению запроса?» Предложение запроса определяет,

соответствует ли ему документ, и вычисляет коэффициент соответствия. По сути, контекст запроса аналогичен запросам SQL, тогда как контекст фильтра аналогичен поиску.

Использование Elasticsearch вместо SQL допускает как поиск, так и запросы, снимая дублирование требований к пространству хранения данных, и устраняет накладные расходы на обслуживание базы данных SQL. Мне встречались сервисы, которые используют только Elasticsearch для хранения данных.

Тем не менее Elasticsearch часто служит не заменой, а дополнением к реляционным базам данных. Это бессхемная база данных, в которой отсутствует концепция нормализации или такие отношения между таблицами, как первичные или внешние ключи. В отличие от SQL, Elasticsearch также не обеспечивает разделения ответственности на команды и запросы (см. раздел 1.4.6) или ACID.

Более того, язык Elasticsearch Query Language (EQL) базируется на JSON, он весьма многословен и сложен в изучении. Язык SQL знаком неразработчикам: аналитикам данных и нетехническому персоналу. Нетехнические пользователи легко могут изучить основы SQL за день.

Язык Elasticsearch SQL был представлен в июне 2018 года с выходом Elasticsearch 6.3.0 (https://www.elastic.co/blog/an-introduction-to-elasticsearch-sql-with-practical-examples-part-1 и https://www.elastic.co/what-is/elasticsearch-sql). Он поддерживает все распространенные операции фильтров и агрегирования (https://www.elastic.co/guide/en/elasticsearch/reference/current/sql-functions.html). Это весьма перспективное направление. SQL продолжает занимать доминирующее положение, но скорее всего, в ближайшие годы все больше сервисов будет использовать Elasticsearch для хранения данных, а также для поиска.

2.6.5. Реализация поиска в сервисах

Упоминание поиска применительно к историям пользователей и обсуждению функциональных требований на собеседованиях демонстрирует ориентацию на клиента. Вам вряд ли придется описывать реализацию поиска помимо создания индексов Elasticsearch, поглощения данных и индексирования, создания поисковых запросов и обработки результатов, если только вопрос не связан с проектированием поискового движка.

В части 2 поиск в основном обсуждается именно на этом уровне.

2.6.6. Дополнительная литература по теме поиска

Дополнительные источники информации об Elasticsearch и индексировании:

- https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html. Официальное руководство по Elasticsearch.

- Madhusudhan Konda. Elasticsearch in Action (Second Edition), Manning Publications, 2023. Практическое руководство по разработке полнофункциональных поисковых систем на базе Elasticsearch и Kibana.
- https://www.manning.com/livevideo/elasticsearch-7-and-elastic-stack. Учебный курс по Elasticsearch 7 и Elastic Stack.
- https://www.manning.com/liveproject/centralized-logging-in-the-cloud-with-elasticsearch-and-kibana. Практический курс ведения журналов в облаке на базе Elasticsearch и Kibana.
- https://stackoverflow.com/questions/33858542/how-to-really-reindex-data-in-elasticsearch. Хорошая альтернатива официальному руководству по Elasticsearch в отношении обновления индексов Elasticsearch.
- https://developers.soundcloud.com/blog/how-to-reindex-1-billion-documents-in-1-hour-at-soundcloud. Пример ситуационного исследования масштабной операции переиндексирования.

2.7. ДРУГИЕ ТЕМЫ

Придя к заключению, что дизайн системы удовлетворяет всем требованиям, можно переходить к другим темам. Вот что еще можно обсудить на собеседовании.

2.7.1. Обслуживание и расширение приложения

Мы начали собеседование с обсуждения требований, а затем на их основе разработали дизайн системы. Можно продолжить работу над дизайном, чтобы он лучше удовлетворял требованиям.

Кроме того, обсуждение можно расширить до других требований. Любой, кто работает в технологической отрасли, знает, что приложения постоянно эволюционируют. Всегда появляются новые и противоречивые требования. Пользователи предоставляют обратную связь о том, что они хотят улучшить или изменить. Мы отслеживаем трафик или содержимое запроса конечных точек API, чтобы принимать обоснованные решения относительно масштабирования и развития. Обсуждение того, какую функциональность необходимо разрабатывать, сопровождать, объявлять устаревшей и выводить из эксплуатации, продолжается непрерывно. Вот примеры тем, которые можно затронуть:

- Возможно, вопросы обслуживания уже рассматривались в ходе собеседования. Какие компоненты системы зависят от технологии (например, программных пакетов), какие разрабатываются быстрее других и требуют наибольших усилий по обслуживанию? Как работать с критическими обновлениями компонентов?
- Функциональность, необходимость в разработке которой может возникнуть в будущем, и ее влияние на дизайн системы.

- Функциональность, которая может стать излишней в будущем, и пути ее безболезненного вывода из эксплуатации. Какой адекватный уровень пользовательской поддержки необходимо обеспечить в этом процессе, как лучше ее предоставить?

2.7.2. Поддержка других типов пользователей

Сервис можно расширить, чтобы он поддерживал другие типы пользователей. Если до этого вы сосредоточились на категориях «потребительские» или «корпоративные», вручную или программно, теперь можно обсудить расширение системы для поддержки других категорий пользователей. Можно также поговорить о расширении текущих сервисов, создании новых и о компромиссах обоих подходов.

2.7.3. Альтернативные архитектурные решения

В начале собеседования, вероятно, обсуждались альтернативные архитектурные решения. Вы можете вернуться к ним и поговорить о них более подробно.

2.7.4. Удобство использования и обратная связь

Удобство использования (юзабилити) оценивает, в какой степени пользователи могут эффективно достигать своих целей с помощью системы и насколько удобно работать с пользовательским интерфейсом. Можно определить метрики юзабилити, сохранять в журнале необходимые данные и реализовать пакетное задание ETL для периодического вычисления этих метрик и обновления дашборда, на котором они выводятся. Метрики юзабилити могут определяться в зависимости от ожиданий относительно того, как пользователи будут работать с системой.

Например, при построении поисковой системы вы хотите, чтобы пользователи быстро находили нужный результат. Одна из возможных метрик — усредненный индекс результата поисковой выдачи, на котором кликает пользователь в списке результатов. Вы упорядочиваете результаты поиска по убыванию соответствия; в этом случае предполагается, что низкий усредненный индекс выбора означает, что результат, интересующий пользователя, находится где-то вверху списка.

Другая возможная метрика — объем помощи, который понадобится пользователям от службы поддержки при использовании приложения. В идеале приложение должно быть самообслуживаемым; иначе говоря, пользователь должен иметь возможность сделать все необходимое прямо в приложении, чтобы ему не приходилось обращаться за помощью. Если у приложения есть техподдержка, метрику можно оценить по количеству заявок, созданных за день или неделю. Большое количество заявок указывает на то, что приложение не является самообслуживаемым.

Удобство использования также можно оценивать по данным опросов пользователей. Стандартная опросная метрика удобства использования — показатель лояльности клиентов, или *NPS (Net Promoter Score)*. NPS определяется как

процент клиентов, оценивающих вероятность того, что они порекомендуют приложение другу или коллеге, на уровне 9 или 10, за вычетом процента поставивших оценку 6 и менее по шкале от 0 до 10.

UI-компоненты для отправки обратной связи можно создавать прямо в приложении. Например, в пользовательском интерфейсе веб-приложения может присутствовать ссылка HTML или форма для отправки обратной связи либо комментариев. Если вы не хотите использовать электронную почту, чтобы это не выглядело как рассылка спама, можно создать конечную точку API для отправки обратной связи и связать ее с отправкой формы.

Хорошая система ведения журналов облегчает воспроизведение ошибок, помогая связать обратную связь пользователя с сохраненной информацией о его действиях.

2.7.5. Граничные случаи и новые ограничения

В конце собеседования эксперт может добавить граничные случаи и новые ограничения — все зависит только от его воображения. Это могут быть новые функциональные требования или экстремальный случай некоторых нефункциональных требований. Возможно, вы уже предвидели какие-то граничные случаи в ходе планирования требований. Обсудите, можно ли пойти на компромиссы для их выполнения или переработать архитектуру для поддержки как текущих, так и новых требований. Несколько примеров:

Новые функциональные требования:

- вы спроектировали сервис продаж с поддержкой платежей по кредитным картам. Что, если платежная система должна поддерживать разные требования к платежам по кредитным картам в разных странах? А если необходимо поддерживать и другие виды платежей, например кредиты, предоставляемые магазином? Или скидочные купоны?
- Вы спроектировали сервис поиска по тексту. Как расширить его для графики, аудио и видео?
- Вы спроектировали сервис бронирования номеров в отелях. А если пользователь захочет сменить номер? Необходимо найти для него подходящий вариант — возможно, в другом отеле.
- А что, если добавить поддержку социальных сетей в рекомендательный сервис каналов новостей?

Масштабируемость и производительность:

- Что, если у пользователя миллион подписчиков или миллион получателей его сообщений? Допустимо ли долгое время доставки сообщений 99-го процентиля? Или необходимо спроектировать более высокопроизводительную систему?

- А если необходимо провести точный аудит данных продаж за последние 10 лет?

Задержка и пропускная способность:

- Что, если время доставки сообщений 99-го процентиля должно лежать в пределах 500 миллисекунд?
- Вы спроектировали сервис видеостриминга, который не поддерживает прямые трансляции. Как изменить дизайн, чтобы сервис их поддерживал? Можно ли обеспечить одновременный стриминг миллиона видеороликов в высоком разрешении на 10 миллиардах устройств?

Доступность и отказоустойчивость:

- Вы спроектировали кэш, который не требовал высокой доступности, потому что все данные также находятся в базе данных. А если понадобится обеспечить высокую доступность по крайней мере для некоторых данных?
- А если сервис продаж использовался для высокочастотного трейдинга? Как повысить его доступность?
- Какие сбои могут происходить в каждом компоненте системы? Как предотвратить или минимизировать последствия этих сбоев?

Расходы:

- Для обеспечения низкой задержки и высокой производительности в ходе проектирования пришлось принять дорогостоящие решения. Чем можно пожертвовать ради снижения расходов?
- Как корректно вывести сервис из эксплуатации в случае необходимости?
- Были ли учтены проблемы портируемости? Как переместить приложение в облако (или из облака)? На какие компромиссы придется пойти, чтобы сделать приложение портируемым? (Более высокие расходы и сложность.) Рассмотрите возможность использования MinIO (https://min.io/) для портируемого хранения объектов.

Каждый вопрос в части 2 книги завершается списком тем для дальнейшего обсуждения.

2.7.6. Облачно-ориентированные концепции

Можно обсудить возможности обеспечения нефункциональных требований через облачно-ориентированные концепции, такие как микросервисы, сервисная сеть и расширения для общих сервисов (Istio), контейнеризация (Docker), оркестрация (Kubernetes), автоматизация (Skaffold, Jenkins), инфраструктура как код (Terraform, Helm). Подробное обсуждение этих тем выходит за рамки

книги. Заинтересованные читатели легко найдут подходящие книги или материалы в интернете.

2.8. АНАЛИЗ ПРОШЕДШЕГО СОБЕСЕДОВАНИЯ

Чем больше собеседований вы пройдете, тем эффективнее они для вас станут. Чтобы вынести как можно больше из каждого собеседования, кратко запишите свои мысли после него, не откладывая. Так у вас появится лучший из возможных протоколов собеседования, и вы сможете честно критически оценить, насколько хорошо вы показали себя.

2.8.1. Запишите свои мысли как можно быстрее

Чтобы вам было проще, в конце собеседования вежливо попросите разрешения сфотографировать свои диаграммы — но не настаивайте, если вам откажут. Принесите с собой ручку и блокнот. Если вам не разрешат фотографировать, постарайтесь перерисовать диаграммы в блокнот. Затем запишите все подробности, которые вам удастся вспомнить.

Всегда записывайте свои мысли как можно быстрее, пока подробности свежи в памяти. Даже если вы устали, не торопитесь расслабляться; вы можете забыть что-то, что поможет повысить эффективность ваших будущих собеседований. Быстро возвращайтесь домой или в номер и запишите свои мысли о собеседовании в комфортной обстановке, чтобы вас ничто не отвлекало.

Структура записей должна быть примерно такой:

1. Заголовок:
   a) Компания и группа, в которой проходило собеседование.
   b) Дата собеседования.
   c) Имя и должность эксперта.
   d) Задача, поставленная экспертом.
   e) Вы сфотографировали диаграммы или восстановили по памяти?
2. Разделите собеседование на примерно 10-минутные части. Поместите диаграммы в тех частях, когда вы начали их рисовать. Фотографии могут содержать несколько диаграмм, так что, возможно, вам придется разделить их.
3. Запишите в разделы как можно больше подробностей, которые вы можете вспомнить.
   a) Что вы сказали.
   b) Что вы нарисовали.
   c) Что сказал эксперт.

4. Запишите свои личные оценки и впечатления. Ваши оценки могут быть неточными, и вы должны стараться улучшать их путем практики.
   a) Попробуйте найти резюме эксперта или профиль на LinkedIn.
   b) Поставьте себя на место эксперта. Как вы думаете, почему эксперт выбрал эту задачу по проектированию системы? Как вы думаете, чего ожидал эксперт?
   c) Выражение и язык тела эксперта. Был ли эксперт удовлетворен вашими суждениями и диаграммами? Перебивал ли вас эксперт или он был готов обсуждать все ваши утверждения? Какие это были утверждения?
5. Если в ближайшие дни вы вспомните что-то еще, запишите это в отдельных разделах, чтобы случайно не внести неточности в исходный отчет.

Пока вы записываете впечатления, задайте себе следующие вопросы:

- Какие вопросы эксперт задавал о вашем дизайне?
- Ставил ли эксперт под сомнение ваши утверждения, например, спрашивал ли он: «А вы уверены?»
- Что собеседник не сказал вам? Как вы думаете, он сделал это намеренно, чтобы посмотреть, упомянете ли вы об этом, или он об этом не знал?

А когда вы закончите записывать свои размышления и воспоминания, отдохните — вы это заслужили.

2.8.2. Оцените себя

Оценка поможет вам понять, в каких областях вы продемонстрировали компетентность или некомпетентность. Оцените себя не позже чем через несколько дней после собеседования.

Запишите все соображения по перечисленным ниже вопросам. Это нужно, чтобы понимать текущие границы своих знаний и свою готовность к следующему собеседованию по проектированию систем.

2.8.3. О чем вы не сказали

Невозможно полностью обсудить систему за 50 минут. Вы сами выбираете, на какие подробности обратить внимание за это время. На основании своих текущих знаний (то есть до начала исследования), как вы думаете, какие подробности вам следовало добавить? Почему вы не упомянули их во время собеседования?

Вы сознательно решили не обсуждать их? Почему? Вы решили, что эти подробности не актуальны или они низкоуровневые? Или были другие причины, по которым вы решили остановиться на других деталях?

Объяснялось ли это нехваткой времени? Можно ли было эффективнее управлять временем собеседования, чтобы его хватило на обсуждение этих тем?

Может, дело в недостатке знаний? Что ж, вы это понимаете. Изучите материал, чтобы увереннее с ним работать.

Вы устали? Не выспались? Может, стоило отдохнуть за день до собеседования, а не пытаться проглотить как можно больше информации? Или вы устали из-за уже прошедших собеседований? Может, стоило попросить устроить короткий перерыв перед собеседованием? Возможно, аромат чашки кофе на столе взбодрил бы вас.

Вы перенервничали? Эксперт или что-то другое заставляли вас чувствовать себя неловко? Поищите в интернете способы, как сохранять спокойствие.

Вас тяготило бремя ожиданий? Старайтесь воспринимать происходящее на перспективу. Есть много хороших компаний. А может, вам повезло, и вы будете приняты в компанию не такую престижную, но обладающую отличными перспективами, так что ваш опыт и вклад будут оценены по достоинству. Вы знаете, что вы настроены учиться каждый день, и что бы ни произошло, это будет лишь один из многих поводов узнать больше, чтобы улучшить результаты будущих собеседований.

Какие подробности были выбраны ошибочно? Их выбор указывает на концепции, которые вы недостаточно хорошо знаете. Изучите материал и эти концепции лучше.

Наконец, поищите ресурсы с информацией по поставленной задаче, например книги или сетевые ресурсы, включая следующие:

- Google.
- Такие веб-сайты, как http://highscalability.com/.
- Видео на YouTube.

Как мы всегда подчеркиваем, задачи по проектированию систем можно решать разными способами. В материалах, которые вы найдете, будет что-то общее, но будут и отличия. Сравните материалы со своими ответами на собеседовании:

- Уточнение задачи. Задавали ли вы правильные вопросы? Что вы упустили из виду?
- Диаграммы. Приводятся ли в материалах понятные блок-схемы? Сравните высокоуровневые архитектурные диаграммы и низкоуровневые диаграммы структуры компонентов со своими.
- Как их высокоуровневая архитектура удовлетворяет этим требованиям? На какие компромиссные решения вам пришлось пойти? Не были ли эти компромиссы слишком дорогостоящими? Какие технологии были выбраны и почему?

- Эффективность коммуникации.
  - Какую часть материала вы поняли с первого раза, когда только прочитали или увидели его?
  - Что осталось непонятным? Объяснялось ли это нехваткой знаний или недостатками изложения? Что можно изменить, чтобы вы поняли информацию с первого раза?

Отвечая на эти вопросы, вы научитесь ясно и лаконично презентовать сложные и нестандартные идеи.

В будущем вы всегда можете дополнить свою оценку. Даже через несколько месяцев после собеседования у вас могут появиться новые мысли, от возможностей улучшения до альтернативных подходов, которые вы могли бы предложить. Извлеките как можно больше пользы из опыта собеседования.

Вы можете (и должны) обсуждать задачи с другими, но никогда не называйте компанию, которая эти задачи ставила. Уважайте конфиденциальность экспертов и чистоту процесса собеседования. Все мы связаны этическими и профессиональными обязательствами по соблюдению единых правил игры, чтобы компании нанимали специалистов по достоинству, а мы могли работать и учиться у других компетентных инженеров. Если все мы будем делать то, что должно, производительность отрасли и наша выгода только возрастут.

2.8.4. Обратная связь по итогам собеседования

Запросите обратную связь по итогам собеседования. Если в компании действует политика непредоставления обратной связи, вам ее не дадут, но спросить никогда не повредит.

Компания, в свою очередь, тоже может запросить у вас обратную связь по электронной почте или телефону. Поделитесь своим мнением, если вас попросят. Помните: даже если ваше мнение не повлияет на решение о найме, вы поможете экспертам просто как коллега-инженер.

2.9. ИНТЕРВЬЮИРОВАНИЕ КОМПАНИИ

Эта книга в основном рассматривает собеседования по проектированию систем с точки зрения соискателя. В этом разделе обсуждается ряд вопросов, которые вы можете задать как соискатель, чтобы решить, готовы ли вы посвятить этой компании следующие несколько лет своей жизни.

Процесс собеседования работает в обе стороны. Компания хочет понять ваш опыт, квалификацию и пригодность, чтобы принять на работу лучшего кандидата из всех имеющихся. Вы проведете в компании по крайней мере несколько лет своей жизни, а значит, вы должны работать с лучшими людьми, а также с лучшими

практиками разработки и философией; это позволит вам максимально развить свои навыки инженера.

Ниже перечислены некоторые способы оценить возможности развития своих инженерных навыков. Перед собеседованием прочитайте технический блог компании. Если статей слишком много, выберите 10 самых популярных или те, которые относятся к вашей вакансии. Из каждой статьи о конкретном инструменте необходимо вынести следующее:

1. Что это за инструмент?
2. Кто им пользуется?
3. Что инструмент делает? Как он это делает? Чем то, что он делает, похоже или отличается от других инструментов? Что он может сделать из того, что не могут сделать другие инструменты? Как он это делает? Что инструмент не может сделать из того, что могут сделать другие инструменты?

Постарайтесь записать не менее двух вопросов по каждому пункту. Перед собеседованием просмотрите свои вопросы и решите, какие из них следует задать во время собеседования.

Что необходимо понять о компании:

- Общий технологический стек компании.
- Инфраструктура и инструменты для работы с данными, используемые в компании.
- Какие инструменты были приобретены, а какие были разработаны? Как принимались эти решения?
- Какие инструменты распространяются с открытым исходным кодом?
- Какой вклад в разработку с открытым исходным кодом внесла компания?
- История и разработка разных инженерных проектов.
- Количество и распределение инженерных ресурсов, выделенных проекту, — вице-президент и директор как наблюдатели проекта, структура, иерархия, опыт и квалификация технических менеджеров, менеджеров проектов и инженеров (фронтенд-инженеров, бэкенд-инженеров, инженеров по работе с данными и data science, инженеров мобильных платформ, инженеров по безопасности и т. д.).
- Статус инструментов. Насколько хорошо разработчики инструментов предвидели потребности пользователей и удовлетворили их? Какие лучшие практики и болевые точки были обнаружены в инструментах компании по результатам частой обратной связи? Разработка каких инструментов была остановлена и почему? Как выглядят эти инструменты в сравнении с инструментами конкурентов и передовыми разработками в этой области?

- Что сделала компания или команды внутри компании для решения этих проблем?
- Какой опыт работы с инструментарием непрерывной интеграции/непрерывной доставки (CI/CD) имеют инженеры компании? Как часто инженеры сталкивались с проблемами CI/CD? Встречались ли случаи, в которых CI-сборки были успешными, но CD-развертывание завершилось неудачей? Сколько времени они потратили на диагностику этих проблем? Сколько сообщений было отправлено в соответствующие службы технической поддержки за последний месяц на одного инженера?
- Какие проекты были запланированы, какие потребности они удовлетворяли? Каково стратегическое видение инженерного отдела?
- Проводились ли миграции в масштабе организации за последние два года? Примеры миграций:
  - Перевод сервисов с физического оборудования на облачного провайдера или между облачными провайдерами.
  - Отказ от использования некоторых инструментов (например, базы данных вроде Cassandra — конкретного решения для мониторинга).
- Происходили ли резкие повороты, например миграция с физического оборудования на облачную платформу Google, а затем миграция на AWS всего через год? В какой степени эти резкие повороты были обусловлены непредсказуемостью, упущениями при планировании или политическими факторами?
- Сколько нарушений в системе безопасности было обнаружено в истории компании, насколько серьезными они были, существует ли риск будущих взломов? Это чувствительный вопрос, и компании открывают эту информацию только по юридическим запросам.
- Общий уровень инженерных компетенций компании.
- Опыт работы руководства — как на текущих, так и на предыдущих ролях.

Особое внимание следует уделить технической подготовке вашего потенциального руководителя. Инженер-программист или технический менеджер никогда не должен принимать нетехнического руководителя, особенно харизматичного. Руководитель, который не может критически оценить работу инженера, не сможет принимать качественные решения в отношении важных изменений в инженерных процессах или руководить внесением таких изменений (например, облачно-ориентированных процессах, таких как переход с ручного развертывания на непрерывное), и может отдавать предпочтение быстрой разработке функциональности за счет накопления технического долга, объема которого он не представляет. Такой руководитель обычно много лет работает в этой компании (или в приобретенной ею), заручился политической поддержкой, которая позволяет ему сохранять свою позицию, и не смог бы

занять аналогичную должность в других компаниях с сильной инженерной структурой. Большие компании, которые продвигают таких менеджеров, проигрывают конкуренцию стартапам. Работа в таких компаниях может быть выгоднее в краткосрочной перспективе, чем имеющиеся альтернативы, но может и препятствовать долгосрочному росту в инженерной сфере. Кроме того, иногда такие вакансии оказываются еще и менее финансово привлекательными, потому что другие компании, которым вы отказали ради краткосрочного финансового выигрыша, по прошествии времени показывают лучшие результаты на рынке, что приводит к быстрому росту вознаграждения для своего персонала. Думайте, прежде чем действовать.

Задайтесь вопросом: что я смогу (и не смогу) узнать в этой компании за следующие четыре года? Когда вы получите оффер, вы сможете проанализировать собранную информацию и принять обоснованное решение.

https://blog.pragmaticengineer.com/reverse-interviewing/ — статья, посвященная собеседованиям с потенциальным руководителем и командой.

ИТОГИ

- Все решения подразумевают компромиссы. Низкая задержка и высокая доступность повышают расходы и сложность. Каждое улучшение одних функций приводит к ухудшению других.
- Не забывайте о времени. Выясните важные пункты обсуждения и сосредоточьтесь на них.
- Начните обсуждение с уточнения требований к системе и анализа компромиссов в характеристиках системы, чтобы оптимизировать ее для соответствия требованиям.
- Затем создайте предварительную спецификацию API для удовлетворения функциональным требованиям.
- Нарисуйте связи между пользователями и данными. Какие данные пользователи читают и записывают в системе и как изменяются данные при перемещении между компонентами системы?
- Обсудите другие возможные темы: ведение журналов, мониторинг, оповещения, поиск и т. д.
- После собеседования проведите письменную самооценку, чтобы лучше понять свои сильные и слабые стороны. Это будет отправная точка для отслеживания вашего будущего прогресса.
- Знайте, чего вы хотите добиться в следующие несколько лет. Проведите интервью с компанией, чтобы определить, стоит ли вкладывать в нее свой опыт.

- Ведение журналов, мониторинг и оповещения важны для быстрого информирования о неожиданных событиях и предоставляют полезную информацию для их обработки.
- Используйте четыре главных сигнала и три инструмента для количественной оценки наблюдаемости сервиса.
- Записи в журнале должно быть легко парсить; они должны быть краткими, содержательными, разделены на категории, использовать стандартизированный формат времени и не должны содержать конфиденциальной информации.
- Применяйте лучшие практики реагирования на оповещения, например полезные и понятные пошаговые руководства. Непрерывно дорабатывайте свои руководства и методы на основании паттернов, которые вы выявили.

3 Нефункциональные требования

В ЭТОЙ ГЛАВЕ

- ✓ Обсуждение нефункциональных требований в начале собеседования
- ✓ Методы и технологии, обеспечивающие соответствие нефункциональным требованиям
- ✓ Оптимизация по нефункциональным требованиям

В системе имеются функциональные и нефункциональные требования. Функциональные требования описывают входные и выходные данные системы. Их можно представить в форме предварительной спецификации API и конечных точек.

Под *нефункциональными требованиями* понимаются те, что не связаны с входными и выходными данными. К числу типичных нефункциональных требований относятся следующие (более подробно мы рассмотрим их далее в этой главе):

- *Масштабируемость* — способность системы эффективно и без повышения затрат регулировать использование аппаратных ресурсов с сохранением текущей нагрузки.
- *Доступность* — процент времени, в течение которого система может принимать запросы и возвращать нужный ответ.

- *Производительность/задержка/99-й процентиль и пропускная способность* — задержкой называется время, затраченное на возвращение системой ответа на запрос пользователя. Максимальная частота, с которой система может обрабатывать запросы, называется ее полосой пропускания (bandwidth). Пропускная способность (throughput) — текущая частота запросов, обрабатываемых системой. Впрочем, на практике очень часто (хотя и неверно) термин «пропускная способность» используется вместо термина «полоса пропускания». Пропускная способность может рассматриваться как величина, обратная задержке. Система с низкой задержкой обладает высокой пропускной способностью.
- *Отказоустойчивость* — способность системы продолжить работу, если в некоторых из ее компонентов происходят сбои, и предотвращение долгосрочного вреда (например, потери данных) при возникновении простоев.
- *Безопасность* — предотвращение несанкционированного доступа к системе.
- *Конфиденциальность* — управление доступом к персональным данным (Personally Identifiable Information, PII), которые могут использоваться для однозначной идентификации пользователя.
- *Точность* — данные системы могут не быть абсолютно точными, и зачастую для уменьшения затрат или сложности требуются компромиссы по точности.
- *Консистентность (согласованность данных)* — соответствие между данными на всех узлах/машинах.
- *Затраты* — для понижения затрат можно пойти на компромиссы в части нефункциональных свойств системы.
- *Сложность, сопровождаемость, отладки и тестирования* — взаимосвязанные концепции, определяющие, насколько трудно будет создать систему, а затем сопровождать ее после завершения.

Клиенты — как технические, так и нетехнические — могут не выражать нефункциональные требования явно, а просто по умолчанию предполагать, что система будет им соответствовать. Следовательно, заявленные требования клиента почти всегда будут неполными, неверными, а иногда избыточными. Без уточнений требования всегда будут вызывать недопонимание. Возможно, какие-то требования не будут сформулированы и они не будут выполняться или же вы будете предполагать наличие некоторых требований, которые на самом деле не нужны, а ваше решение будет избыточным.

Новичок скорее допустит ошибки при уточнении нефункциональных требований, но недостаточное уточнение возможно как для функциональных, так и для нефункциональных. Обсуждение проектирования любой системы должно начинаться с обсуждения и уточнения обоих видов требований.

Нефункциональные требования часто противоречат друг другу. На любом собеседовании, касающемся проектирования систем, необходимо обсудить, какие решения помогут достичь компромиссов.

Обсуждать нефункциональные требования отдельно от методов их реализации достаточно сложно, поскольку некоторые методы обеспечивают выигрыш в одних требованиях, но потери в других. В оставшейся части главы мы кратко обсудим все нефункциональные требования и некоторые методы их выполнения, а затем рассмотрим каждый метод более подробно.

3.1. МАСШТАБИРУЕМОСТЬ

Масштабируемость (scalability) — способность системы эффективно и без повышения затрат регулировать использование аппаратных ресурсов с сохранением текущей нагрузки.

Процесс расширения для поддержания большей нагрузки или количества пользователей называется *масштабированием*. Масштабирование требует увеличения вычислительной мощности процессора, оперативной памяти, пространства хранения данных и пропускной способности сети. Под термином «масштабирование» может пониматься как вертикальное, так и горизонтальное масштабирование.

Вертикальное масштабирование концептуально прямолинейно; его легко обеспечить простым увеличением денежных затрат. Оно подразумевает переход на более мощный и дорогостоящий хост с более быстрым процессором, бóльшим объемом памяти, бóльшим жестким диском, SSD вместо механического жесткого диска для снижения задержки либо сетевым адаптером с большей пропускной способностью. У вертикального масштабирования есть три основных недостатка.

Во-первых, вскоре достигается точка, в которой денежные затраты растут быстрее производительностя обновленного оборудования. Например, специализированный мейнфрейм с несколькими процессорами будет стоить больше, чем эквивалентное количество отдельных серийных серверов, каждый из которых оснащен одним процессором.

Во-вторых, у вертикального масштабирования имеются технологические пределы. Независимо от бюджета текущие технологические ограничения определяют максимальный объем вычислительной мощности, оперативной памяти или пространства хранения данных, возможные на одном хосте.

В-третьих, вертикальное масштабирование связано с простоями. Следует остановить хост, заменить оборудование, а затем запустить снова. Чтобы избежать простоев, необходимо подготовить другой хост, запустить на нем сервис, а затем организовать перенаправление запросов на новый хост. Но это возможно только в том случае, если состояние сервиса будет храниться на другой машине, отличной от старого или нового хоста. Как будет показано далее, для обеспечения соответствия многим нефункциональным требованиям — масштабируемости, доступности, отказоустойчивости — применяются такие методы, как перенаправление запросов конкретным хостам или хранение состояния сервиса на другом хосте.

Вертикальное масштабирование как концепция достаточно очевидно, поэтому далее в книге под терминами «масштабируемый» и «масштабирование» следует понимать «горизонтально масштабируемый» и «горизонтальное масштабирование» (если явно не указано обратное).

Горизонтальное масштабирование означает распределение требований к обработке и хранению данных по многим хостам. «Настоящая» масштабируемость обеспечивается только горизонтально. Горинзонтальное масштабирование почти всегда обсуждается на собеседованиях по проектированию систем.

Требования к масштабируемости определяются на основании следующих вопросов:

- Сколько данных поступает в систему и возвращается системой?
- Сколько выполняется запросов на чтение в секунду?
- Какой объем данных связан с каждым запросом?
- Сколько просмотров видео должно поддерживаться в секунду?
- Насколько большими могут быть внезапные выбросы трафика?

3.1.1. Сервисы с сохранением и без сохранения состояния

HTTP относится к категории протоколов без сохранения состояния, поэтому использующий его бэкенд-сервис легко масштабируется горизонтально. В главе 4 обсуждается горизонтальное масштабирование чтения из базы данных. Бэкенд-сервис на базе HTTP без сохранения состояния в сочетании с горизонтально масштабируемыми операциями чтения из базы данных — хорошая отправная точка для обсуждения масштабируемого дизайна систем.

Больше всего трудностей возникает с масштабированием операций записи в общее хранилище. Далее в книге мы обсудим разные методы, включая репликацию, сжатие, агрегирование, денормализацию и сервис метаданных.

За обсуждением распространенных коммуникационных архитектур, включая плюсы и минусы решений с сохранением и без сохранения состояния, обращайтесь к разделу 6.7.

3.1.2. Основные концепции балансировщиков нагрузки

Каждый горизонтально масштабируемый сервис использует балансировщик нагрузки. Возможны следующие варианты:

- Аппаратный балансировщик нагрузки — специализированное физическое устройство, распределяющее трафик между несколькими хостами. Аппаратные балансировщики нагрузки известны своей дороговизной (от нескольких тысяч до сотен тысяч долларов).

- Общий сервис балансировщика нагрузки, также обозначаемый сокращением LBaaS (Load Balancing as a Service, «балансировщик нагрузки как сервис»).
- Сервер с установленным ПО балансировщика нагрузки. Самые популярные решения — HAProxy и NGINX.

В этом разделе рассматриваются основные концепции балансировщиков нагрузки, которые могут встретиться в ходе собеседования. На диаграммах, приведенных в книге, я обозначаю различные сервисы или другие компоненты прямоугольниками, а запросы — стрелками. Обычно предполагается, что запросы к сервису проходят через балансировщик нагрузки и маршрутизируются к хостам сервиса. Сами балансировщики нагрузки на диаграммах обычно не обозначаются.

Вы можете сказать эксперту, что изображать компонент балансировщика нагрузки на системных диаграммах не обязательно, так как его присутствие подразумевается, а его изображение и обсуждение отвлекает от других составных частей сервиса.

Уровень 4 и уровень 7

Вы должны уметь различать балансировщики нагрузки уровня 4 и уровня 7[1] и объяснить, какой из них лучше подходит для каждого конкретного сервиса. Балансировщик нагрузки уровня 4 работает на транспортном уровне (TCP). Он принимает решения о маршрутизации на основании адресной информации, извлекаемой из нескольких первых пакетов в потоке TCP, и не анализирует содержимое других пакетов; он умеет только перенаправлять пакеты. Балансировщик нагрузки уровня 7 работает на уровне приложения (HTTP) и поэтому обладает следующими возможностями:

- *Решения по распределению нагрузки/маршрутизации* — принимаются на основании содержимого пакетов.
- *Аутентификация* — может вернуть код 401 при отсутствии заданного заголовка аутентификации.
- *Завершение TLS-запросов* — требования безопасности к трафику в датацентре могут быть ниже, чем требования к трафику в интернете, и применение завершения TLS-запросов (HTTPS → HTTP) означает отсутствие затрат на шифрование/дешифрование между хостами датацентра. Если приложение требует, чтобы трафик в датацентре шифровался (шифрование при передаче), завершение TLS-запросов не применяется.

Липкая сессия

Термином «липкая сессия» (sticky session) называется отправка запросов балансировщиком нагрузки от конкретного клиента конкретному хосту в течение

[1] В рамках семиуровневой сетевой модели OSI (см. https://ru.wikipedia.org/wiki/8C_OSI). — *Примеч. ред.*

периода времени, определяемого балансировщиком нагрузки или приложением. Липкие сессии используются для сервисов с сохранением состояния. Например, сайт интернет-магазина, социальной сети или банка может использовать липкие сессии для сохранения данных пользовательского сеанса (например, информации входа или предпочтений в профиле), чтобы пользователю не приходилось заново проходить аутентификацию или вводить свои предпочтения при навигации по сайту. Сайт интернет-магазина может использовать липкие сессии для хранения содержимого корзины.

Липкую сессию можно реализовать с использованием cookie-файлов, основанных на сроке действия или управляемых приложением. В сессии на базе срока действия балансировщик нагрузки выдает клиенту значение cookie, определяющее продолжительность сессии. Каждый раз, когда балансировщик нагрузки получает запрос, он проверяет cookie. В сессии под управлением приложения cookie генерируется приложением. Балансировщик нагрузки все еще генерирует собственное значение cookie поверх cookie, выданного приложением, но cookie-файл балансировщика нагрузки подчиняется сроку действия cookie-файла приложения. Такой подход гарантирует, что клиенты не будут перенаправляться другому хосту после истечения срока действия cookie балансировщика нагрузки, но он сложнее в реализации, поскольку требует дополнительной интеграции между приложением и балансировщиком нагрузки.

Репликация сессии

При *репликации сессии* операции записи в хост копируются в другие хосты кластера, назначенные той же сессии, так что операции чтения могут маршрутизироваться к любому хосту в сессии. Таким образом улучшается доступность.

Такие хосты могут образовать кольцо резервирования. Например, если в сессии задействованы три хоста и в хост A выполняется запись, данные записываются в хост B, который, в свою очередь, выполняет запись в хост C. Другой способ основан на выдаче запросов на запись балансировщиком нагрузки ко всем хостам, назначенным сессии.

Балансировщик нагрузки и обратные прокси-серверы

Термин «обратное проксирование» также может встречаться в подготовительных материалах для собеседований по проектированию систем. Мы кратко сравним балансировку нагрузки с обратным проксированием.

Балансировка нагрузки ориентирована на масштабируемость, тогда как обратное проксирование является методом управления коммуникацией «клиент — сервер». Обратный прокси-сервер располагается перед кластером серверов и выполняет функции шлюза между клиентами и серверами, перехватывая и перенаправляя входящие запросы соответствующему серверу на основании

URL и других критериев. Обратный прокси-сервер также может предоставлять средства повышения производительности (такие, как кэширование и сжатие) и средства безопасности (такие, как завершение SSL). Балансировщики нагрузки также могут предоставлять завершение SSL, но их главной целью является масштабируемость.

Сравнение балансировки нагрузки и обратного проксирования представлено по адресу https://www.nginx.com/resources/glossary/reverse-proxy-vs-load-balancer/.

Дополнительная литература

- https://www.cloudlare.com/learning/performance/types-of-load-balancing-algorithms/ — краткое описание различных алгоритмов балансировки нагрузки.
- https://rancher.com/load-balancing-in-kubernetes — хорошее введение в тему балансировки нагрузки в Kubernetes.
- https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer и https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/ — описание присоединения балансировщика нагрузки внешнего облачного сервиса к сервису Kubernetes.

3.2. ДОСТУПНОСТЬ

Доступность (availability) — процент времени, в течение которого система может принимать запросы и возвращать нужный ответ. Стандартные целевые показатели доступности приведены в табл. 3.1.

Таблица 3.1. Стандартные целевые показатели доступности

Доступность, %	Время простоя за год	Время простоя за месяц	Время простоя за неделю	Время простоя за день
99,9 («три девятки»)	8,77 часа	43,8 минуты	10,1 минуты	1,44 минуты
99,99 («четыре девятки»)	52,6 минуты	4,38 минуты	1,01 минуты	8,64 секунды
99,999 («пять девяток»)	5,26 минуты	26,3 секунды	6,05 секунды	864 миллисекунды

За подробным обсуждением мультирегиональной конфигурации Netflix со всеми активными узлами для обеспечения высокой доступности обращайтесь к https://netflixtechblog.com/active-active-for-multi-regional-resiliency-c47719f6685b. В этой книге рассматриваются похожие методы для обеспечения высокой доступности, например репликация внутри датацентров и между датацентрами на разных континентах. Также будут рассмотрены темы мониторинга и оповещений.

Высокая доступность необходима для большинства сервисов, и другими нефункциональными требованиями можно пожертвовать для обеспечения высокой доступности без излишней сложности.

При обсуждении нефункциональных требований системы сначала установите, нужна ли высокая доступность. Не стоит полагать, что строгая согласованность и низкая задержка обязательны. Обратитесь к теореме CAP и обсудите, можно ли пожертвовать ими ради высокой доступности. По возможности предложите использовать асинхронные средства коммуникаций, которые достигают этой цели, такие как порождение событий (event sourcing) и саги (saga), рассматриваемые в главах 4 и 5.

Сервисы, запросы которых не нуждаются в немедленной обработке, а ответы возвращаются немедленно — такие, как запросы, осуществляемые на программном уровне между сервисами, вряд ли потребуют строгой согласованности и низкой задержки. Примеры — сохранение журнала в долгосрочном хранилище данных или отправка запроса на бронирование в Airbnb. Синхронные коммуникационные протоколы обычно используются для запросов, отправляемых непосредственными пользователями приложения, когда на них абсолютно необходим немедленный ответ.

Тем не менее не стоит предполагать, что на все запросы, отправляемые людьми, необходимо немедленно выдавать ответ с запрашиваемыми данными. Подумайте, достаточно ли для ответа выдать подтверждение, а запрашиваемые данные вернуть через несколько минут или часов. Например, если пользователь отправляет запрос на оплату подоходного налога, платеж не обязан пройти немедленно. Сервис может поставить запрос во внутреннюю очередь, а пользователю выдать немедленный ответ о том, что запрос будет обработан в течение нескольких минут или часов. Позднее платеж может быть обработан потоковым заданием или периодическим пакетным заданием, после чего пользователя можно оповестить о результате (например, был ли платеж выполнен успешно или отклонен) по таким каналам, как электронная почта, смс-сообщения или уведомления в приложениях.

Пример ситуации, в которой высокая доступность может оказаться необязательной, встречается в сервисе кэширования. Так как кэширование может использоваться для сокращения задержки и сетевого трафика запроса и не обязательно для выполнения самого запроса, можно пожертвовать доступностью для уменьшения задержки в дизайне сервиса кэширования. Другой пример — ограничение частоты — обсуждается в главе 8.

Для измерения доступности также можно использовать метрики инцидентов. В статье https://www.atlassian.com/incident-management/kpis/common-metrics описываются такие метрики инцидентов, как MTTR (Mean Time to Recovery, «среднее время восстановления») и MTBF (Mean Time Between Failures, «среднее время безотказной работы»). Часто для этих метрик создаются свои дашборды и сигналы.

3.3. ОТКАЗОУСТОЙЧИВОСТЬ

Отказоустойчивостью (fault-tolerance) называется способность системы продолжать работу при сбое некоторых из ее компонентов. Она способствует корректному снижению функциональности, когда при отказе некоторых частей система частично сохраняет функциональность вместо полного критического отказа. Таким образом, у инженеров появляется время для исправления отказов и восстановления работоспособности системы. Также можно реализовать механизмы самовосстановления, которые автоматически предоставляют компоненты-заменители и присоединяют их к системе, чтобы она могла восстанавливаться без ручного вмешательства и без заметного эффекта для конечного пользователя.

Темы доступности и отказоустойчивости часто обсуждаются вместе. Хотя доступность является метрикой исправности/простоя, отказоустойчивость — это не метрика, а характеристика системы.

С темой отказоустойчивости тесно связана концепция отказоустойчивого проектирования, основанного на бесшовной обработке ошибок. Подумайте, как обрабатывать ошибки в сторонних API, которыми вы не управляете, а также скрытые/необнаруженные ошибки. Ниже описаны некоторые средства отказоустойчивости.

3.3.1. Репликация и избыточность

Репликация подробно рассматривается в главе 4.

Один из методов репликации заключается в создании нескольких (например, трех) избыточных экземпляров/копий компонента, чтобы один или два экземпляра могли одновременно отказать без последствий для работоспособности системы. Как обсуждается в главе 4, операции обновления обычно связываются с конкретным хостом, так что производительность обновления пострадает только в том случае, если другие хосты находятся в разных датацентрах, географически удаленных от источника запроса. Но операции чтения часто выполняются на всех репликах, так что эффективность чтения снижается при отказе компонентов.

Один экземпляр назначается источником достоверных данных (он часто называется лидером), а два других — репликами (или последователями). Существуют различные варианты конфигурации реплик. Одна реплика размещается на другой серверной стойке в том же датацентре, а другая реплика — в другом датацентре. В другой конфигурации все три экземпляра размещаются в разных датацентрах, что позволяет максимизировать отказоустойчивость за счет снижения производительности.

Примером служит система HDFS (Hadoop Distributed File System) с настраиваемым свойством, которое называется «коэффициентом репликации» и задает

количество копий любого блока. Значение по умолчанию равно 3. Репликация также помогает повысить доступность.

3.3.2. Прямая коррекция ошибок и коды коррекции ошибок

Прямая коррекция ошибок (FEC, Forward Error Correction) — метод предотвращения ошибок в зашумленных или ненадежных каналах связи, основанный на кодировании сообщений с использованием избыточности, например *кодов коррекции ошибок* (ECC, Error Correction Code).

FEC является концепцией уровня протокола, а не системного уровня. Во время собеседований по проектированию систем вы можете дать понять, что знакомы с понятиями FEC и ECC, но вряд ли дело дойдет до их подробного обсуждения, поэтому в книге они подробно не рассматриваются.

3.3.3. Предохранитель

Предохранитель (circuit breaker) — механизм, не позволяющий клиенту многократно повторять операцию, которая с большой вероятностью завершится неудачей. В контексте нисходящих (downstream) сервисов предохранитель вычисляет количество запросов, завершившихся неудачей в границах недавнего периода времени. При превышении порога ошибок клиент перестает обращаться с вызовами к нисходящим сервисам. Позже клиент попытается выполнить ограниченное количество запросов. Если они будут успешными, клиент предположит, что сбой устранен, и продолжит отправлять запросы без ограничений.

> **ОПРЕДЕЛЕНИЕ** Если сервис B зависит от сервиса A, то сервис A называется восходящим, а B — нисходящим.

Предохранитель экономит ресурсы, которые были бы потрачены на отправку запросов с большой вероятностью отказа. Он также не позволяет клиентам создавать дополнительную нагрузку на уже перегруженную систему.

Однако предохранитель затрудняет тестирование системы. Представим нагрузочный тест, который выдает неверные запросы, но при этом должным образом тестирует предельные параметры системы. Теперь тест активирует предохранитель, и нисходящие сервисы теперь успешно справляются с нагрузкой, которая раньше была для них избыточной. Аналогичная нагрузка со стороны клиентов вызовет сбой. Также затрудняется оценка соответствующих порогов ошибок и таймеров.

Предохранитель может быть реализован на стороне сервера. Пример такого решения — Resilience4j (https://github.com/resilience4j/resilience4j). Образцом для него послужил продукт Hystrix (https://github.com/Netflix/Hystrix), разработанный Netflix

и переведенный в режим поддержки в 2017 году (https://github.com/Netflix/Hystrix/issues/1876#issuecomment-440065505). Вместо заранее настроенных конфигураций, таких как адаптивные пределы конкурентности (https://netflixtechblog.medium.com/performance-under-load-3e6fa9a60581), Netflix сосредоточился на адаптивных реализациях, реагирующих на производительность приложения в реальном времени.

3.3.4. Экспоненциальная задержка с повтором

Экспоненциальная задержка с повтором очень похожа на предохранитель. Когда клиент получает ответ с ошибкой, он некоторое время ожидает, прежде чем повторять попытку, и время ожидания между попытками растет экспоненциально. Клиент также корректирует период ожидания на небольшую случайную отрицательную или положительную величину — этот прием называется «джиттером» (jitter). Он предотвращает отправку запросов несколькими клиентами точно в одно время с возникновением «лавины повторных попыток», которая может перегрузить нисходящий сервис. По аналогии с предохранителем, когда клиент получает ответ с успешным результатом, он предполагает, что сбой устранен, и продолжает отправлять запросы без ограничений.

3.3.5. Кэширование ответов других сервисов

Сервис может зависеть от внешних сервисов для получения некоторых данных. Как поступать в случае недоступности внешнего сервиса? Общепринятый способ — организовать корректное снижение функциональности вместо сбоя или возвращения ошибки. Вместо возвращаемого значения можно использовать значение по умолчанию или пустой ответ. Если устаревшие данные лучше, чем полное отсутствие данных, можно кэшировать ответы внешнего сервиса при выдаче успешных запросов и использовать эти ответы в случае недоступности внешнего сервиса.

3.3.6. Контрольные точки

Машина может выполнять определенные операции агрегирования данных по многим точкам данных. Для этого она систематически выбирает их подмножество, выполняет с ним агрегирование, а затем записывает результат в заданное место. Процесс повторяется, пока все точки данных не будут обработаны, или до бесконечности, как в случае потокового пайплайна. Если на этой машине произойдет сбой во время агрегирования данных, заменяющая машина должна знать, с каких точек данных должно восстановиться агрегирование. Для этого можно сохранять контрольную точку после обработки каждого подмножества точек данных и успешной записи результата. Заменяющая машина может продолжить обработку с контрольной точки.

Контрольные точки обычно используются с пайплайнами ETL, которые используют брокеры сообщений (например, Kafka). Машина может получить несколько

событий из топика Kafka, обработать события, а затем записать результат и создать контрольную точку. Если на машине произойдет сбой, то заменяющая машина может продолжить обработку с последней контрольной точки.

Kafka предоставляет возможность хранения смещений на уровне разделов (https://kafka.apache.org/22/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html). Flink потребляет данные из топиков Kafka и периодически проверяет контрольные точки с использованием механизма распределенных контрольных точек Flink (https://ci.apache.org/projects/link/link-docs-master/docs/dev/datastream/fault-tolerance/checkpointing/).

3.3.7. Очередь недоставленных сообщений

Если запрос на запись к стороннему API завершается неудачей, его можно поставить в очередь недоставленных сообщений и повторить выдачу позже.

Как хранятся очереди недоставленных сообщений — локально или в удаленном сервисе? Можно поискать баланс между сложностью и удобочитаемостью:

- Простейший вариант: если пропуск запросов допустим, удалить ошибочные запросы.
- Локальная реализация очереди недоставленных сообщений с блоком `try-catch`. В случае сбоя хоста запросы будут потеряны.
- Более сложный и надежный вариант — использование потоковой событийной платформы (например, Kafka).

На собеседовании следует обсудить разные подходы с их плюсами и минусами. Не ограничивайтесь одним подходом.

3.3.8. Ведение журналов и периодический аудит

Один из методов обработки скрытых ошибок основан на регистрации запросов на запись в журналах и выполнении периодического аудита. Задание аудита может обрабатывать журналы и проверять, что данные сервиса, в который выполняется запись, совпадают с ожидаемыми значениями. Эта тема более подробно обсуждается в главе 10.

3.3.9. Паттерн Bulkhead

Паттерн Bulkhead («переборка»[1]) описывает механизм отказоустойчивости, при котором система делится на изолированные пулы, чтобы сбой в одном пуле не влиял на всю систему.

---

[1] Название происходит из морской индустрии, где переборки используются для изоляции отсеков корабля. — *Примеч. ред.*

Например, разные конечные точки сервиса могут иметь собственные пулы потоков вместо совместного использования пула потоков, поэтому если пул потоков конечной точки будет исчерпан, это не повлияет на способность других конечных точек к обслуживанию запросов (за дополнительной информацией обращайтесь к книге Индразири и Сиривардена (Indrasiri and Siriwardena) «Microservices for the Enterprise: Designing, Developing, and Deploying» (Apress, 2019).

Другой пример паттерна рассматривается в книге Майкла Нейгарда (Michael T. Nygard) «Release It!: Design and Deploy Production-Ready Software»[1] (Pragmatic Bookshelf, 2018). Запрос может привести к отказу хоста из-за бага. При каждом повторении этого запроса выходит из строя очередной хост. Разбиение сервиса на изолированные части (то есть разбиение хостов на пулы) не позволит этому запросу вывести из строя все хосты и вызвать полную неработоспособность. Запрос необходимо проанализировать, так что в сервисе должно быть предусмотрено ведение журналов и мониторинг. Мониторинг обнаруживает вредоносный запрос, а инженеры могут использовать журналы для диагностики сбоев и определения их причин.

Источник запроса может выдавать запросы к сервису с высокой частотой, не позволяя последнему обслуживать другие источники запросов. Паттерн Bulkhead связывает некоторые хосты с определенным источником запросов, не позволяя ему потреблять всю емкость сервиса. (Ограничение частоты, рассматриваемое в главе 8, — еще один способ предотвращения таких ситуаций.)

Хосты сервиса можно разделить на пулы и связать с каждым пулом источники запросов. Этот же прием используется для назначения приоритетов некоторым источникам запросов с выделением им дополнительных ресурсов.

На рис. 3.1 сервис обслуживает два других сервиса. Недоступность хостов сервиса не позволит ему обслуживать другие источники запросов.

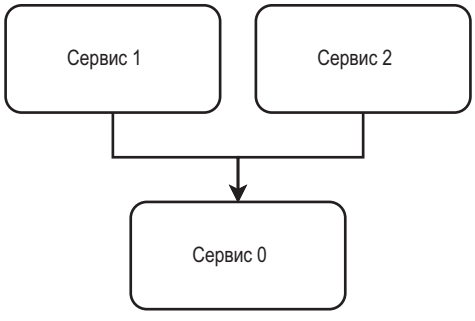

Рис. 3.1. Все запросы к сервису 0 распределяются между хостами. Недоступность хостов сервиса 0 не позволит ему обслуживать другие источники запросов

[1] Нейгард М. «Release it! Проектирование и дизайн ПО для тех, кому не всё равно». СПб., издательство «Питер».

На рис. 3.2 хосты сервиса разделены на пулы, которые связываются с источниками запросов. Недоступность хостов одного пула не отразится на других источниках. Очевидный недостаток такого решения заключается в том, что пулы не смогут поддерживать друг друга при возникновении выбросов трафика от некоторых источников. Это намеренное решение, которое мы принимаем для выделения определенного количества хостов конкретному источнику запросов. Пулы могут масштабироваться по мере необходимости — как вручную, так и автоматически.

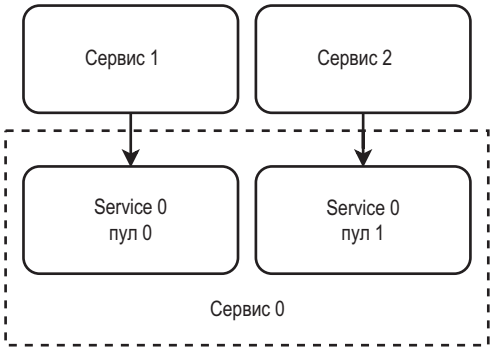

Рис. 3.2. Сервис 0 делится на два пула, каждый из которых связывается с источником запросов. Недоступность одного пула не влияет на другой

За другими примерами использования паттерна Bulkhead обращайтесь к книге Майкла Нейгарда «Release It!: Design and Deploy Production-Ready Software».

Мы не будем упоминать паттерн Bulkhead при обсуждении проектирования систем в части 2, но он обычно применяется в большинстве систем, и вы можете обсудить его в ходе собеседования.

3.3.10. Паттерн Fallback

Паттерн Fallback («резервный вариант») включает выявление проблемы и последующее выполнение альтернативного кода, например кэшированных ответов или альтернативных сервисов, сходных с сервисом, от которого клиент пытается получить информацию. Например, если клиент запрашивает у бэкенда список ближайших кафе с домашней выпечкой, ответ можно кэшировать для использования в будущем, если внутренний сервис столкнется со сбоем. Кэшированный ответ может содержать устаревшую информацию, но это все же лучше для пользователя, чем сообщение об ошибке. Клиент также может отправить запрос к стороннему картографическому API (такому, как Bing или Google Maps), на котором может не оказаться специализированного контента, предоставляемого нашим бэкендом. Проектируя резервный вариант, следует учитывать его надежность и то, что в нем тоже может произойти сбой.

ПРИМЕЧАНИЕ В статье https://aws.amazon.com/builders-library/avoiding-fallback-in-distributed-systems/ вы найдете дополнительную информацию о стратегиях резервных решений, поймете, почему Amazon почти никогда не применяет этот паттерн, а также узнаете про альтернативы, используемые Amazon.

3.4. ЗАДЕРЖКА И ПРОПУСКНАЯ СПОСОБНОСТЬ

Задержкой (latency) называется время, необходимое для возвращения ответа системой на запрос пользователя. Задержка включает сетевую задержку передачи запроса от клиента к сервису, время, необходимое сервису для обработки запроса и создания ответа, а также сетевую задержку возвращения запроса от сервиса к клиенту. У типичного запроса потребительского приложения (например, просмотра ресторанного меню в приложении или проведения платежа в приложении интернет-магазина) задержка должна составлять от десятков миллисекунд до нескольких секунд. В приложениях для высокочастотного трейдинга продолжительность задержки может составлять несколько миллисекунд.

Строго говоря, под задержкой понимается время перемещения пакета от источника к приемнику. Тем не менее термин «задержка» часто считают синонимом термина «производительность». Для обсуждения времени перемещения пакета мы будем использовать термин «задержка».

Термин «задержка» также может использоваться для описания времени «запрос — ответ» между компонентами внутри системы (а не для времени «запрос — ответ» для пользовательских запросов). Например, если внутренний хост выдает запрос к подсистеме ведения журналов или хранения данных, задержка будет равна времени, необходимому для ведения журналов / хранения данных и возвращения ответа внутреннему хосту.

Функциональные требования системы могут означать, что ответ может и не содержать информации, запрашиваемой пользователем, а быть простым подтверждением наряду с обещанием того, что после истечения заданного интервала запрашиваемая информация будет отправлена пользователю или пользователь сможет получить ее, выдав другой запрос. Такой компромисс может упростить дизайн системы, поэтому всегда стоит уточнять требования и обсуждать, через какое время после запроса пользователю потребуется информация.

Рассмотрим несколько типичных проектировочных решений для достижения низкой задержки. Можно развернуть сервис в датацентре, географически приближенном к пользователям, чтобы пакетам не приходилось преодолевать большие расстояния между пользователями и сервисом. Если пользователи имеют большой географический разброс, сервис можно развернуть в нескольких датацентрах, выбранных с учетом минимизации расстояния до кластера пользователей. Если хостам между датацентрами нужен совместный доступ к данным, сервис должен быть горизонтально масштабируемым.

Иногда могут присутствовать другие факторы, которые влияют на задержку сильнее, чем физическое расстояние между пользователями и датацентрами, — например трафик, или пропускная способность сети, или объем работы бэкенда (реальная бизнес-логика и уровень долгосрочного хранения данных). Можно организовать передачу тестовых запросов между пользователями и датацентрами и определить датацентр с наименьшей задержкой для пользователей из конкретного региона.

Существуют и другие методы: CDN, кэширование, уменьшение размера данных за счет использования RPC вместо REST, разработка собственного протокола с таким фреймворком, как Netty, для использования TCP и UDP вместо HTTP, а также пакетные и потоковые технологии.

Обсуждение задержки включает рассмотрение характеристик данных и путей поступления данных в систему и вывода из нее, после чего можно предлагать соответствующие стратегии. Можно ли подсчитывать просмотры уже через несколько часов? В зависимости от ответа на этот вопрос будут применяться пакетные или потоковые решения. Каким должно быть время ответа? Если небольшим, то данные должны агрегироваться и агрегирование должно выполняться во время записи, а во время чтения быть минимальным или не проводиться вообще.

3.5. КОНСИСТЕНТНОСТЬ

Консистентность, или *согласованность* (consistency), имеет разное значение в ACID и CAP. Консистентность ACID основывается на отношениях данных, таких как внешние ключи и уникальность. Как утверждается в книге Мартина Клеппмана (Martin Kleppmann) «Designing Data-Intensive Applications»[1] (O'Reilly, 2017), консистентность CAP в действительности представляет собой линеаризуемость, то есть все узлы, содержащие одинаковые данные на некоторый момент времени, и изменения в данных должны быть линейными; другими словами, узлы должны начать проводить изменения одновременно.

Консистентностью баз данных в конечном счете жертвуют ради улучшения доступности, масштабирования и задержки. Базы данных ACID, включая РСУБД, не принимают операции записи при нарушении связности сети, потому что в таком случае они не могут обеспечить консистентность ACID. Как показано в табл. 3.2, в MongoDB, HBase и Redis на первый план выходит линеаризуемость в ущерб доступности, тогда как в CouchDB, Cassandra, Dynamo, Hadoop и Riak ситуация противоположная.

---

[1] Клеппман М. «Высоконагруженные приложения. Программирование, масштабирование, поддержка». СПб., издательство «Питер».

Таблица 3.2. Предпочитаемые характеристики баз данных

Линеаризуемость	Доступность
HBase	Cassandra
MongoDB	CouchDB
Redis	Dynamo
	Hadoop
	Riak

В ходе обсуждения следует подчеркнуть различия между консистентностью ACID и CAP и компромиссы между линеаризуемостью и консистентностью в конечном счете. В этой книге рассматриваются различные методы обеспечения линеаризуемости и консистентности в конечном счете, включая следующие:

- полносвязная сеть (full mesh);
- кворум.

Методы консистентности в конечном счете, основанные на записи в одном месте, от которого запись распространяется в другие места:

- порождение событий (раздел 5.2) — метод обработки выбросов трафика;
- сервис координации;
- распределенный кэш.

Методы консистентности в конечном счете, жертвующие консистентностью и точностью для снижения затрат:

- gossip-протокол;
- случайный выбор лидера.

Некоторые недостатки линеаризуемости:

- снижение доступности, так как большинство узлов должно быть уверено в консенсусе, прежде чем они смогут обслуживать запросы. Проблема усугубляется с ростом количества узлов;
- повышение сложности и затрат.

3.5.1. Полносвязная сеть

На рис. 3.3 представлен пример полносвязной сети. Каждый хост в кластере знает адреса всех остальных хостов и рассылает им сообщения.

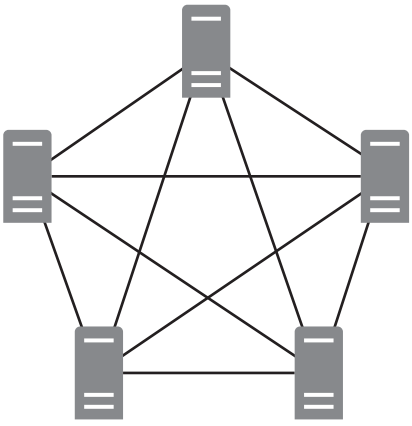

Рис. 3.3. Схема полносвязной сети. Каждый хост соединен со всеми остальными хостами и направляет им сообщения

Как хосты обнаруживают друг друга? Как при добавлении нового хоста его адрес рассылается другим хостам? Ниже перечислены некоторые решения по обнаружению хостов:

- Ведение списка адресов в файле конфигурации. При каждом изменении списка файл развертывается на всех хостах/узлах.
- Использование стороннего сервиса, прослушивающего контрольные сообщения всех хостов. Хост остается зарегистрированным, пока сервис получает контрольные сообщения. Все хосты используют этот сервис для получения полного списка адресов.

Полносвязная сеть реализуется проще других методов, но плохо масштабируется. Количество сообщений растет в квадратичной зависимости от количества хостов. Полносвязная сеть хорошо работает в малых кластерах, но не способна обеспечивать работоспособность больших кластеров. В стратегии кворума, чтобы система считалась согласованной, достаточно того, чтобы большинство хостов располагало одинаковыми данными. BitTorrent — пример протокола, использующего полносвязную сеть для децентрализованного совместного доступа к файлам по схеме p2p. Во время собеседования можно кратко упомянуть о полносвязной сети и сравнить ее с масштабируемыми решениями.

3.5.2. Сервис координации

На рис. 3.4 представлен сервис координации — сторонний компонент, который выбирает лидирующий узел или набор лидирующих узлов. Наличие лидера сокращает количество сообщений. Все остальные узлы отправляют свои сообщения лидеру, а лидер может выполнить необходимую обработку и вернуть окончательный результат. Каждому узлу достаточно связаться с лидером или набором лидеров, а каждый лидер управляет набором узлов.

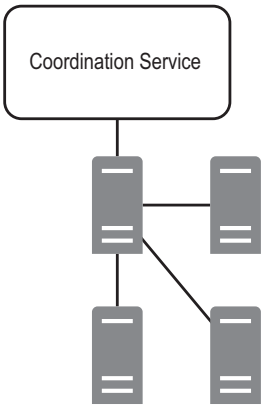

Рис. 3.4. Сервис координации

Примеры алгоритмов координации — Paxos, Raft и Zab. Другим примером служит метод «один лидер, несколько последователей» в SQL (раздел 4.3.2), допускающий масштабируемое чтение. ZooKeeper (https://zookeeper.apache.org/) — распределенный сервис координации. ZooKeeper обладает рядом преимуществ перед хранением файла конфигурации на одном хосте (многие из них обсуждаются в статье по адресу https://stackoverlow.com/q/36312640/1045085). Эту функциональность можно реализовать в распределенной файловой системе или распределенной базе данных, но ZooKeeper уже предоставляет ее:

- Управление доступом (https://zookeeper.apache.org/doc/r3.1.2/zookeeperProgrammers.html#sc_ZooKeeperAccessControl).
- Хранение данных в памяти для высокой производительности.
- Горизонтальная масштабируемость посредством добавления хостов в ZooKeeper Ensemble (https://zookeeper.apache.org/doc/r3.1.2/zookeeperAdmin.html#sc_zkMulitServerSetup).
- Гарантированная консистентность в конечном счете в пределах заданного времени или сильная консистентность с более высокими затратами (https://zookeeper.apache.org/doc/current/zookeeperInternals.html#sc_consistency).
- ZooKeeper жертвует доступностью ради консистентности; это CP-система в теореме CAP.
- Клиенты могут читать данные в порядке их записи.

Главный недостаток сервиса координации — сложность. Сервис координации представляет собой сложно организованный компонент, который должен быть высоконадежным и гарантировать, что будет выбран один и только один лидер. (Ситуация, в которой два узла полагают, что каждый из них является лидером, называется «расщеплением сознания» (split brain); подробности см. в книге Мартина Клеппмана «Designing Data-Intensive Applications».)

3.5.3. Распределенное кэширование

Также можно воспользоваться распределенным кэшем, таким как Redis или Memcached. На рис. 3.5 узлы сервиса направляют периодические запросы к источнику для загрузки новых данных, а затем — запросы к распределенному кэшу (например, к хранилищу в памяти, такому как Redis) для обновления данных. Это простое решение с низкой задержкой, а кластер распределенного кэша может масштабироваться независимо от сервиса. Однако такое решение использует больше запросов, чем все остальные, которые мы описали, кроме полноячеистой сети.

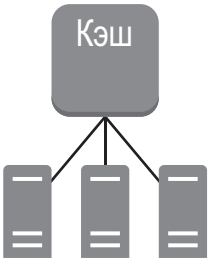

Рис. 3.5. Использование распределенного кэша для рассылки сообщений. Узлы могут выдавать запросы к хранилищу в памяти (например, Redis) для обновления данных или же выдавать периодические запросы для выборки новых данных

ПРИМЕЧАНИЕ Redis представляет собой кэш в памяти и не является строго распределенным по определению. Однако его механизмы кластеризации способны удовлетворять практические запросы систем. Подробнее см. https://redis.io/docs/about/ and https://stacoverlow.com/questions/18376665/redis-distributed-or-not.

Как хост-отправитель, так и хост-получатель могут убедиться, что сообщение содержит все необходимые поля. Часто это делают обе стороны, поскольку при незначительных дополнительных затратах такая проверка сокращает вероятность ошибок, из-за которых все сообщение становится недействительным. Когда хост-отправитель отправляет недействительное сообщение хосту-получателю по запросу HTTP, а хост-получатель может обнаружить, что сообщение недействительно, он может немедленно вернуть код 400 или 422. Можно задать критические оповещения, срабатывающие по ошибкам 4xx, чтобы немедленно узнавать об ошибке и заняться ее исследованием. Однако при использовании Redis недействительные данные, записанные узлом, могут остаться необнаруженными, пока не будут прочитаны другим узлом, так что оповещения могут приходить с задержкой.

Запросы, отправленные напрямую с одного хоста другому хосту, проходят проверку схемы. Однако Redis — всего лишь база данных, которая не проверяет схему, и хосты могут записывать в нее произвольные данные. Это может создать проблемы с безопасностью (см. https://www.trendmicro.com/en_us/research/20/d/exposed-redis-instances-abused-for-remote-code-execution-cryptocurrency-mining.html и https://www.imperva.com/blog/new-research-shows-75-of-open-redis-servers-infected).

База данных Redis проектировалась для обращений со стороны доверенных клиентов внутри безопасных сред (https://redis.io/topics/security). Redis не поддерживает шифрование, что может создать проблемы с безопасностью. Реализация шифрования в клиенте повышает сложность и затраты, а также снижает производительность (https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/at-rest-encryption.html). Сервис координации преодолевает эти недостатки, но повышает сложность и затраты.

3.5.4. Gossip-протокол

Gossip-протокол моделирует распространение эпидемий. Как показано на рис. 3.6, каждый узел выбирает другой случайный узел (периодически или со случайным интервалом), а затем совместно использует данные. Такой протокол жертвует согласованностью в угоду снижению затрат и сложности.

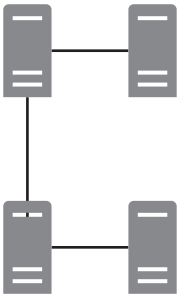

Рис. 3.6. Gossip-коммуникация

Cassandra использует gossip-протокол для обеспечения согласованности между распределенными секциями данных. В DynamoDB используется gossip-протокол, называемый vector clocks, для поддержания согласованности между многими датацентрами.

3.5.5. Случайный выбор лидера

На рис. 3.7 используется простой алгоритм случайного выбора лидера. Этот алгоритм не гарантирует, что лидер будет только один, а значит, их может быть несколько. Это несущественно, потому что каждый лидер может обмениваться данными со всеми остальными хостами, поэтому все хосты, включая всех лидеров, будут иметь верные данные. Недостатки — риск дублирования запросов и избыточного сетевого трафика.

Kafka использует модель репликации «лидер — последователь» со случайным выбором лидера для обеспечения отказоустойчивости. YARN использует метод случайного выбора лидера для управления распределением ресурсов в кластере хостов.

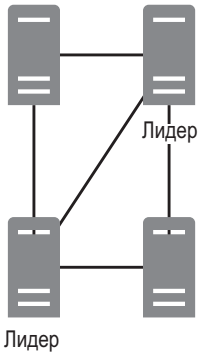

Рис. 3.7. Схема с несколькими лидерами, которая может возникнуть в результате случайного выбора лидера

3.6. ТОЧНОСТЬ

Точность (accuracy) является актуальным нефункциональным требованием в системах со сложной обработкой данных или высокой частотой записи. Точность данных означает, что значения данных верны и не являются приближенными. Оценочные алгоритмы жертвуют точностью ради снижения сложности. Примеры оценочных алгоритмов — HyperLogLog для оценки количества элементов (`COUNT DISTINCT`) в ядре распределенных запросов SQL Presto и Count-min sketch для оценки частот событий в потоке данных.

Кэш устаревает при изменении данных в нижележащей базе данных. У кэша может существовать политика обновления, в соответствии с которой он производит выборку новейших данных в фиксированные интервалы времени. Политики обновления с короткими интервалами более затратны. Возможная альтернатива — обновление или удаление связанного ключа кэша при изменении данных — повышает сложность.

Точность отчасти связана с консистентностью. Системы с консистентностью в конечном счете жертвуют точностью ради повышения доступности и снижения сложности и затрат. При записи в систему с консистентностью в конечном счете результаты чтения, выполненного после записи, могут не отражать эффекты записи, то есть быть неточными. Система с консистентностью в конечном счете остается неточной до тех пор, пока реплики не будут обновлены эффектами операции записи. Однако применительно к таким ситуациям используется термин «консистентность», а не «точность».

3.7. СЛОЖНОСТЬ И СОПРОВОЖДАЕМОСТЬ

Первый шаг к минимизации *сложности* (complexity) — прояснить как функциональные, так и нефункциональные требования, чтобы не проектировать лишнего.

При составлении структурных диаграмм обращайте внимание на то, какие компоненты могут быть выделены в независимые системы. Используйте общие сервисы для сокращения сложности и повышения удобства обслуживания. Типичные примеры сервисов, обобщаемых практически для любых ситуаций:

- сервис распределения нагрузки;
- ограничение частоты (см. главу 8);
- аутентификация и авторизация (см. приложение Б);
- ведение журналов, мониторинг, оповещения (см. раздел 2.5);
- завершение TLS-запросов. За дополнительной информацией обращайтесь к другим источникам;
- кэширование (см. раздел 4.8);
- DevOps и CI/CD (если уместно). Эти темы выходят за рамки книги.

Сервисы, обобщаемые для некоторых организаций (например, сервисы, собирающие данные пользователей для анализа), включают аналитику и машинное обучение.

Сложные системы могут потребовать еще большей сложности для обеспечения высокой доступности и высокой отказоустойчивости. Если сложность понизить нельзя, постарайтесь найти баланс между сложностью и доступностью/отказоустойчивостью.

Обсудите возможные компромиссы в части других требований для снижения сложности, например пайплайны ETL для отложенной обработки данных, которые не обязательно обрабатывать в реальном времени.

Стандартный способ снижения задержки и повышения производительности за счет сложности основан на использовании методов, минимизирующих размер сообщений в сетевых коммуникациях. К числу таких методов относятся фреймворки сериализации RPC и сервисы метаданных (за информацией о сервисах метаданных обращайтесь к разделу 6.3).

Фреймворки сериализации RPC, такие как Avro, Thrift и protobuf, позволяют сокращать размер сообщений за счет обслуживания файлов схем. (За сравнением REST с RPC обращайтесь к разделу 6.7.) Всегда предлагайте использовать такие фреймворки сериализации на любом собеседовании (мы не будем больше останавливаться на этом).

Также следует обсудить пути возникновения отказов, оценить эффект отказов для пользователей и бизнеса и способы предотвращения отказов и восстановления. Среди стандартных концепций стоит упомянуть репликацию, отработку отказов и составление документации (см. раздел 2.5.3).

Сложность будет обсуждаться во всех главах части 2.

3.7.1. Непрерывное развертывание (CD)

Непрерывное развертывание (CD) впервые упоминалось в книге в разделе 1.4.5. В частности, обсуждалось, что CD упрощает развертывание и откат. При этом образуется быстрый цикл обратной связи, упрощающий обслуживание системы. Если вы случайно запустите в эксплуатацию сборку с ошибками, ее будет легко откатить. Быстрые и простые развертывания инкрементных обновлений и новой функциональности приводят к ускорению жизненного цикла разработки. В этом заключается главное преимущество сервисов перед монолитными архитектурами (см. приложение А).

Также к числу методов CD относятся «сине-зеленые» развертывания, иначе называемые развертываниями с нулевыми простоями. За подробностями обращайтесь к таким источникам, как https://spring.io/blog/2016/05/31/zero-downtime-deployment-with-a-database, https://dzone.com/articles/zero-downtime-deployment, и https://craftquest.io/articles/what-are-zero-downtime-atomic-deployments.

Такие инструменты статического анализа кода, как SonarQube (https://www.sonarqube.org/), также улучшают сопровождаемость системы.

3.8. ЗАТРАТЫ

При обсуждении дизайна систем нужно понимать, что выгоднее — повысить затраты или пожертвовать другими нефункциональными требованиями. Примеры:

- Повышение затрат ради снижения сложности за счет применения вертикального масштабирования (вместо горизонтального).
- Снижение доступности ради уменьшения затрат за счет снижения избыточности системы (например, количества хостов или коэффициента репликации в базе данных).
- Повышение задержки для уменьшения затрат за счет использования датацентра в более экономичном месте, находящемся дальше от пользователей.

Обсудите затраты на реализацию и мониторинг и затраты по каждому нефункциональному требованию (например, высокой доступности).

Проблемы в процессе эксплуатации различаются по серьезности и по тому, насколько быстро их необходимо решать; следовательно, не реализуйте больше мониторинга и оповещений, чем необходимо. Затраты повышаются, если инженеры должны узнавать о проблеме сразу же после ее возникновения (по сравнению с допустимостью оповещения через несколько часов после инцидента).

Помимо затрат на обслуживание в форме решения возможных проблем на продакшен, существуют затраты, обусловленные естественной амортизацией

программного продукта со временем, когда библиотеки и сервисы начинают устаревать. Найдите компоненты, которым могут понадобиться будущие обновления. Какие зависимости (например, библиотеки) помешают простому обновлению компонентов, если перестанут поддерживаться? Как спроектировать систему, чтобы упростить обновление этих зависимостей при необходимости?

Насколько вероятно, что вам придется изменять зависимости в будущем, особенно сторонние зависимости, которыми вы не управляете? Сторонние зависимости могут быть выведены из эксплуатации или не удовлетворять вашим требованиям (например, к надежности или безопасности).

Полное обсуждение затрат должно учитывать затраты на вывод системы из эксплуатации в случае необходимости. Это может произойти по разным причинам, например, команда решает сменить сферу деятельности или системе не хватает пользователей для оправдания затрат на ее разработку и обслуживание. Возможно, вы захотите предоставить существующим пользователям доступ к их данным; тогда эти данные потребуется извлечь в текстовые и/или CSV-файлы.

3.9. БЕЗОПАСНОСТЬ

В ходе собеседования иногда приходится обсуждать уязвимости в системах, а также возможности предотвращения и снижения угроз безопасности. К этой категории относятся как обращения извне, так и обращения внутри организации. В отношении безопасности обычно обсуждаются следующие темы:

- Завершение TLS-запросов или передача шифрованных данных между сервисами или хостами в датацентре (*шифрование при передаче*). Завершение TLS-запросов обычно применяется для экономии на обработке, поскольку шифрование между хостами в датацентре обычно не требуется. Возможны исключения для конфиденциальных данных, для которых может применяться шифрование при передаче.
- Какие данные могут храниться без шифрования, а какие должны храниться в зашифрованном виде (*шифрование при хранении*). Шифрование при хранении обычно концептуально отличается от хранения хешированных данных.

Желательно хотя бы немного разбираться в OAuth 2.0 и OpenID Connect, описанных в приложении Б.

Также можно обсудить ограничение частоты запросов для предотвращения атак DDoS. Система ограничения частоты запросов может стать отдельным вопросом на собеседовании (эта тема обсуждается в главе 8). О ней стоит упоминать при проектировании практически любых систем, ориентированных на внешнюю сеть.

3.10. КОНФИДЕНЦИАЛЬНОСТЬ

Персональные данные (Personally Identifiable Information, PII) — это данные, которые могут использоваться для однозначной идентификации клиента: полное имя, почтовый адрес, адреса электронной почты и идентификаторы банковских счетов. Персональные данные должны быть защищены для соответствия таким нормативным требованиям, как GDPR (General Data Protection Regulation) и CCPA (California Consumer Privacy Act). Это относится как к внешнему, так и к внутреннему доступу.

В рамках системы механизмы управления доступом должны применяться к персональным данным, хранящимся в базах данных и файлах. Можно использовать такие механизмы, как LDAP (Lightweight Directory Access Protocol). Данные могут шифроваться как при передаче (с использованием SSL), так и при хранении.

Рассмотрите возможность применения алгоритмов хеширования (например, SHA-2 и SHA-3) для маскировки персональных данных и поддержания конфиденциальности отдельных клиентов при вычислении агрегатной статистики (например, среднего количества транзакций на клиента).

Если персональные данные хранятся в базе данных, поддерживающей только добавление данных, или в таких файловых системах, как HDFS, распространенный метод обеспечения безопасности данных заключается в назначении каждому клиенту ключа шифрования. Ключи шифрования могут храниться в изменяемой системе хранения, например в SQL. Данные, связанные с конкретным клиентом, должны шифроваться с его персональным ключом перед сохранением. Если данные клиента потребуется удалить, достаточно удалить ключ шифрования клиента. Тогда все данные клиента в хранилище, поддерживающем только добавление данных, становятся недоступными, то есть фактически удаляются.

Можно обсудить сложность, затраты и влияние конфиденциальности, а также многое другое, включая организацию поддержки, персонализацию и применение машинного обучения.

Кроме того, стоит упомянуть о стратегиях предотвращения и снижения риска несанкционированного доступа к данным, например политики хранения данных и аудита. Эти особенности обычно в каждой организации индивидуальны, так что их обсуждение будет ситуативным.

3.10.1. Внутренние и внешние сервисы

При проектировании внешнего сервиса определенно стоит уделить внимание механизмам безопасности и конфиденциальности. Как насчет внутренних сервисов, которые обслуживают только другие внутренние сервисы? Можно положиться на механизмы защиты пользовательских сервисов от внешних атак и предположить, что внутренние пользователи не будут совершать противоправные

действия, так что для ограничительного сервиса меры безопасности не нужны. Также можно довериться пользовательским сервисам и предположить, что они не будут запрашивать у других пользовательских сервисов данные об источниках запросов, так что меры безопасности не обязательны.

Однако, скорее всего, для надлежащей реализации механизмов безопасности компания не должна полагаться на то, что внутренние пользователи не попытаются (случайно или злонамеренно) нарушить конфиденциальность клиентов. Следует соблюдать культуру реализации механизмов безопасности и конфиденциальности по умолчанию. Этот принцип согласуется с внутренними политиками управления доступом и конфиденциальности для всех видов сервисов и данных, принятых в большинстве организаций. Например, во многих организациях применяется ролевое управление доступом к репозиториям Git и CI/CD всех сервисов. Во многих организациях также устанавливаются процедуры, предоставляющие доступ к данным работников и клиентов только тем лицам, для которых такой доступ считается необходимым. Средства управления доступом и доступ к данным обычно максимально ограничиваются по объему и продолжительности. Нет никаких причин принимать такие политики для одних систем, не принимая их для других. Прежде чем решить, что для внутреннего сервиса можно не реализовывать механизмы безопасности и конфиденциальности, следует убедиться, что он не передает никакие конфиденциальные функции или данные. Более того, все сервисы — как внешние, так и внутренние — должны регистрировать обращения к базам данных, содержащим конфиденциальную информацию.

Другой механизм конфиденциальности заключается в создании четко определенной политики хранения пользовательской информации. Базы данных с пользовательской информацией должны находиться за сервисами, имеющими подробную документацию и жесткие требования к безопасности и управлению доступом. Другие сервисы и базы данных должны хранить только идентификаторы пользователей и никакие другие пользовательские данные. Идентификаторы пользователей необходимо менять либо периодически, либо в случае нарушения безопасности или конфиденциальности.

На рис. 1.8 в главе 1 изображена сервисная сеть с механизмами безопасности и конфиденциальности, представленная в виде внешнего запроса к сервису управления идентификационными данными и доступом.

3.11. ОБЛАЧНО-ОРИЕНТИРОВАННЫЕ РЕШЕНИЯ

Облачно-ориентированные решения — подход к удовлетворению нефункциональных требований, включая масштабируемость, отказоустойчивость и сопровождаемость. Ниже приведено определение облачно-ориентированных решений от Cloud Native Computing Foundation (https://github.com/cncf/toc/blob/main/DEFINITION.md). Я выделил курсивом некоторые ключевые слова:

- Облачно-ориентированные технологии дают возможность организациям строить и выполнять масштабированные приложения в современных динамических средах, таких как общедоступные, приватные и гибридные облака. Этот подход воплощен в *контейнерах*, *сервисных сетях, микросервисах*, *неизменяемых инфраструктурах* и *декларативных API*.
- Все перечисленные методы позволяют создавать *слабосвязанные системы*, которые являются *гибкими*, *управляемыми* и *наблюдаемыми*. В сочетании с мощной автоматизацией они позволяют инженерам вносить высокорезультативные *изменения часто* и *предсказуемо* с минимальными усилиями.
- Cloud Native Computing Foundation стремится содействовать принятию этой парадигмы за счет культивации и обслуживания экосистемы коммерчески нейтральных проектов с открытым исходным кодом. Мы демократизируем современные паттерны, чтобы эти инновации стали доступными для всех.

Облачно-ориентированные вычисления не являются темой этой книги, но мы используем облачно-ориентированные методы (контейнеры, сервисные сети, микросервисы, бессерверные функции, неизменяемую инфраструктуру или «инфраструктуру как код», декларативные API, автоматизацию) для получения преимуществ (гибкость, управляемость, наблюдаемость, возможность частых и предсказуемых изменений). Также книга содержит ссылки на материалы по соответствующим концепциям.

3.12. ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

Заинтересованные читатели могут поискать информацию о теореме PACELC, которая в книге не обсуждается. PACELC является расширением теоремы CAP. Теорема PACELC утверждает, что при нарушении целостности сети в распределенной системе приходится выбирать между доступностью и согласованностью, а в ходе нормальной работы — между задержкой и согласованностью.

Полезный ресурс, по содержанию сходный с этой главой, — книга «Microservices for the Enterprise: Designing, Developing, and Deploying» (2018, Apress), написанная Касуном Индрасири (Kasun Indrasiri) и Прабатом Сириварденой (Prabath Siriwardena).

ИТОГИ

- Необходимо обсудить как функциональные, так и нефункциональные требования системы. Не делайте предположений о нефункциональных требованиях. Можно искать компромиссы между нефункциональными характеристиками для оптимизации таких требований.
- Масштабируемость — способность системы эффективно и без повышения затрат регулировать использование аппаратных ресурсов с сохранением

текущей нагрузки. Эта тема обсуждается практически всегда, поскольку уровень трафика в системе предсказать трудно или невозможно.

- Доступность — процент времени, в течение которого система может принимать запросы и возвращать нужный ответ. Многие — но не все — запросы требуют высокой доступности, поэтому необходимо уточнить, является ли доступность обязательным требованием к системе.
- Отказоустойчивостью называется способность системы продолжать работу при сбое некоторых из ее компонентов и предотвращение непоправимого вреда при простоях. У пользователей остается возможность использования некоторых функций, а у инженеров появляется время для исправления компонентов со сбоем и восстановления работоспособности системы.
- Задержкой называется время, необходимое для возвращения системой ответа на запрос пользователя. Пользователи ожидают, что интерактивные приложения будут быстро загружаться и быстро реагировать на их ввод.
- Консистентность означает, что все узлы в некоторый момент времени содержат одинаковые данные, а при изменении в данных все узлы должны начать проводить изменения одновременно. В некоторых системах (например, финансовых) все пользователи, просматривающие одни и те же данные, должны видеть одинаковые значения, тогда как в других (например, в социальных сетях) допустимо, чтобы разные пользователи видели немного различающиеся данные в любой момент времени — при условии, что в конечном счете данные будут одинаковыми.
- Системы с консистентностью в конечном счете жертвуют точностью для снижения сложности и затрат.
- Сложность должна быть минимальной, чтобы систему было дешевле и проще создавать и обслуживать. По возможности используйте такие общепринятые методы, как общие сервисы.
- Обсуждение затрат должно включать минимизацию сложности, затраты при сбоях, затраты на обслуживание, затраты на переход на другие технологии и затраты на вывод из эксплуатации.
- При обсуждении безопасности необходимо проанализировать, какие данные должны быть защищены, а для каких данных защита не обязательна, а также учесть такие концепции, как шифрование при передаче и шифрование при хранении.
- Вопросы конфиденциальности включают механизмы и процедуры управления доступом, удаления или маскировки пользовательских данных, предотвращения и снижения риска несанкционированного доступа к данным.
- Облачно-ориентированный подход к проектированию системы предполагает ряд методов для удовлетворения нефункциональных требований.

4 Масштабирование баз данных

В ЭТОЙ ГЛАВЕ

- ✓ Разные типы сервисов хранения данных
- ✓ Репликация баз данных
- ✓ Агрегирование событий для снижения количества записей в базу данных
- ✓ Нормализация и денормализация
- ✓ Кэширование частых запросов в памяти

В этой главе обсуждаются концепции масштабирования баз данных (БД), их плюсы и минусы, а также популярные базы данных, использующие эти концепции в реализации. Эти концепции учитываются при выборе БД для различных сервисов системы.

4.1. КРАТКОЕ ВВЕДЕНИЕ В СЕРВИСЫ ХРАНЕНИЯ ДАННЫХ

Сервисы хранения данных имеют состояние. По сравнению с сервисами без сохранения состояния *сервисы с состоянием* имеют механизмы для обеспечения консистентности и требуют избыточности для предотвращения потери данных.

Сервис с состоянием может выбрать такие механизмы, как Paxos, для обеспечения сильной консистентности или консистентности в конечном счете. Это сложные решения, и в ходе проектирования приходится идти на компромиссы, зависящие от разных требований: консистентности, сложности, безопасности, задержки, производительности и т. д. Это одна из причин, по которым все сервисы по возможности реализуются без состояния, а состояние поддерживается только в сервисах с состоянием.

> **ПРИМЕЧАНИЕ** При сильной консистентности все обращения видны всем параллельным процессам (либо узлам, процессорам и т. д.) в одинаковом порядке (последовательно). Следовательно, наблюдаться может только одно согласованное состояние в отличие от слабой консистентности, при которой разные параллельные процессы (или узлы и т. д.) могут воспринимать переменные в разных состояниях.

Другая причина заключается в том, что при хранении состояния на отдельных хостах веб-сервиса или бэкенд-сервиса необходимо реализовать липкие сессии, последовательно направляя одного пользователя на один и тот же хост. Также необходимо реплицировать данные на случай, если на хосте произойдет сбой, и обработать резервный сценарий (например, маршрутизировать пользователя на новый хост в случае сбоя на его хосте). Сохраняя все состояние в сервисе хранения данных с состоянием, мы можем выбрать технологию хранения / базу данных, наиболее подходящую для наших требований, и избежать необходимости проектировать собственный механизм управления состоянием, а также избежать возможных сопутствующих ошибок.

Хранение данных можно приблизительно разбить на несколько категорий, которые нужно уметь различать. Полное описание типов хранения данных выходит за рамки этой книги (при необходимости обращайтесь к другим источникам), а ниже мы перечислим основные моменты, необходимые для понимания вопросов, обсуждаемых в ней:

- *Базы данных*:
  - *SQL* — обладают реляционными характеристиками: включают таблицы и отношения между таблицами, в том числе первичные и внешние ключи. База данных SQL должна обладать свойствами ACID.
  - *NoSQL* — базы данных, не обладающие всеми свойствами SQL.
  - *Столбцово-ориентированные* — данные упорядочиваются по столбцам (а не по строкам) для эффективной фильтрации. Примеры — Cassandra и HBase.
  - *Ключ — значение* — данные хранятся в виде набора пар «ключ — значение». Каждый ключ соответствует области диска, определяемой алгоритмом хеширования. При этом достигается хорошая производительность чтения. Ключи должны быть хешируемыми, то есть представлять собой примитивные типы, а не указатели на объекты. У значений такого ограничения

нет; они могут быть как примитивами, так и указателями. Базы данных «ключ — значение» обычно используются для кэширования с применением таких принципов, как LRU (Least Recently Used). Кэш обладает высокой производительностью, но не требует высокой доступности (если кэш недоступен, источник запросов может обратиться с запросом к источнику данных). Примеры — Memcached и Redis.

- *Документные* — могут интерпретироваться как базы данных «ключ — значение», где значения не имеют ограничений размера или других значимых ограничений как у баз данных «ключ — значение». Значения могут храниться в разных форматах, самые распространенные — текст, JSON и YAML. Пример — MongoDB.
- *Графовые* — проектируются для эффективного хранения отношений между сущностями. Примеры — Neo4j, RedisGraph и Amazon Neptune.
- *Файловые* — данные хранятся в файлах, которые могут быть упорядочены по каталогам/файлам. Структуру можно рассматривать в форме «ключ — значение», где путь является ключом.
- *Блоковое хранилище* — данные хранятся в блоках одинакового размера с уникальными идентификаторами. Блоковые хранилища в веб-приложениях практически не используются. Они актуальны для проектирования низкоуровневых компонентов других систем хранения (например, баз данных).
- *Объектное хранилище* — более плоская иерархия по сравнению с файловым хранилищем. Для обращения к объектам обычно используются простые HTTP API. Запись объектов происходит медленно, и объекты не могут изменяться, так что хранилище объектов подходит для статических данных. Пример облачного хранения — AWS S3.

4.2. КОГДА ИСПОЛЬЗОВАТЬ БАЗЫ ДАННЫХ

Принимая решение о том, как хранить данные сервиса, можно выбирать базы данных или другие средства, например файловые, блоковые и объектные хранилища. В ходе собеседования помните, что хотя вам могут нравиться определенные решения и вы можете высказать свои предпочтения, вы должны быть способны обсудить все важные факторы и принять во внимание мнение других. В этом разделе речь пойдет о темах, которые могут быть затронуты на собеседовании. Как обычно, обговаривайте плюсы и минусы каждого решения.

Выбор между базой данных и файловой системой обычно основывается на личных предпочтениях и опыте. В этой области мало академических исследований или жестких принципов. В часто цитируемом заключении старой статьи Microsoft от 2006 года (https://www.microsoft.com/en-us/research/publication/to-blob-or-not-to-blob-large-object-storage-in-a-database-or-a-filesystem) говорится: «Объекты менее 256 Кбайт лучше хранить в базах данных, а объекты, размер которых превышает

1 Мбайт, лучше хранить в файловых системах. Между 256 Кбайт и 1 Мбайт важными факторами становятся соотношение “чтение:запись” и частота перезаписи или замены объектов». Еще несколько важных моментов:

- SQL Server требует специальных настроек конфигурации для хранения файлов более 2 Гбайт.
- Объекты баз данных полностью загружаются в память, поэтому читать файлы из базы данных неэффективно.
- Если строки таблицы базы данных содержат большие объекты, репликация будет медленной, потому что большие объекты необходимо будет реплицировать из узла-лидера на узел-последователь.

4.3. РЕПЛИКАЦИЯ

Базы данных масштабируются (то есть распределяются по нескольким хостам, которые в терминологии баз данных обычно называются узлами) посредством репликации, секционирования и шардирования. Репликация создает копии данных, называемые репликами, и хранит их на разных узлах. Секционирование (partitioning) и шардирование (sharding) направлены на разбиение набора данных на подмножества. Шардирование подразумевает распределение подмножеств по разным узлам, тогда как при секционировании этого не происходит. У одного хоста имеются ограничения, поэтому он не соответствует нашим требованиям:

- *Отказоустойчивость* — каждый узел может создавать резервные копии своих данных на других узлах в том же или других датацентрах на случай сбоя узла или сети. Можно добавить процесс автоматической отработки отказа, при которой другие узлы берут на себя роли и секции/шарды узлов со сбоем.
- *Повышение емкости хранения* — один узел может вертикально масштабироваться с установкой нескольких жестких дисков наибольшей доступной емкости, но такое решение потребует значительных расходов, и могут возникнуть проблемы с пропускной способностью узла.
- *Повышение пропускной способности* — база данных должна обрабатывать операции чтения и записи для нескольких одновременных процессов и пользователей. Вертикальное масштабирование ограничено возможностями самого быстрого сетевого адаптера, мощностью процессора и объемом памяти.
- *Снижение задержки* — реплики можно распределять географически, чтобы они были ближе к пользователям. Можно увеличить количество определенных реплик в датацентре при увеличении числа операций чтения данных в этом регионе.

Чтобы масштабировать операции чтения (`SELECT`), мы просто увеличиваем количество реплик этих данных. С масштабированием записи дело обстоит сложнее,

и большая часть этой главы посвящена решению проблем с масштабированием операций записи.

4.3.1. Распределение реплик

В типичном дизайне создается одна резервная копия на хосте на той же стойке и еще одна — на хосте на другой стойке и/или в другом датацентре. На эту тему написано довольно много (например, https://learn.microsoft.com/en-us/azure/availability-zones/az-overview).

Также возможно шардирование данных, его преимущества перечислены ниже. Главным компромиссом шардирования становится повышение сложности из-за необходимости отслеживать местоположение шардов:

- *Масштабирование хранилища* — если база данных / таблица слишком велика и не помещается на одном узле, шардирование по узлам позволяет ей сохранить единую логику.
- *Масштабирование памяти* — если база данных хранится в памяти, возможно, ее придется шардировать, поскольку вертикальное масштабирование памяти на одном узле быстро становится слишком дорогим.
- *Масштабирование обработки* — шардированная база данных может использовать возможности параллельной обработки.
- *Локальность* — базу данных можно шардировать так, чтобы данные, необходимые конкретному узлу кластеров, хранились локально, а не в другом шарде другого узла.

ПРИМЕЧАНИЕ В целях линеаризуемости некоторые шардированные базы данных (такие, как HDFS) реализуют удаление как операцию присоединения («логическое обратимое удаление»). Такой подход предотвращает несогласованность операций чтения, которые продолжают выполняться в момент удаления.

4.3.2. Репликация с одним лидером

В репликации с одним лидером все операции записи выполняются на одном узле, который называется лидером. Репликация с одним лидером предназначена для масштабирования чтения, а не записи. У некоторых баз данных SQL, таких как MySQL и Postgres, — имеются конфигурация для репликации с одним лидером. Сервис SQL утрачивает свою согласованность ACID. Это важно, если вы решаете провести горизонтальное масштабирование базы данных SQL для обслуживания сервиса с высоким трафиком.

На рис. 4.1 изображена схема репликации с одним лидером с отработкой отказа лидера по схеме «первичный-вторичный». Все операции записи (также называемые запросами Data Manipulation Language, или запросами DDL, в SQL)

выполняются на узле первичного лидера и реплицируются на его последователях, в том числе на вторичном лидере. Если на первичном лидере происходит сбой, то в ходе отработки отказа вторичный лидер берет на себя роль первичного. Когда работоспособность лидера, на котором произошел сбой, восстановлена, он становится вторичным лидером.

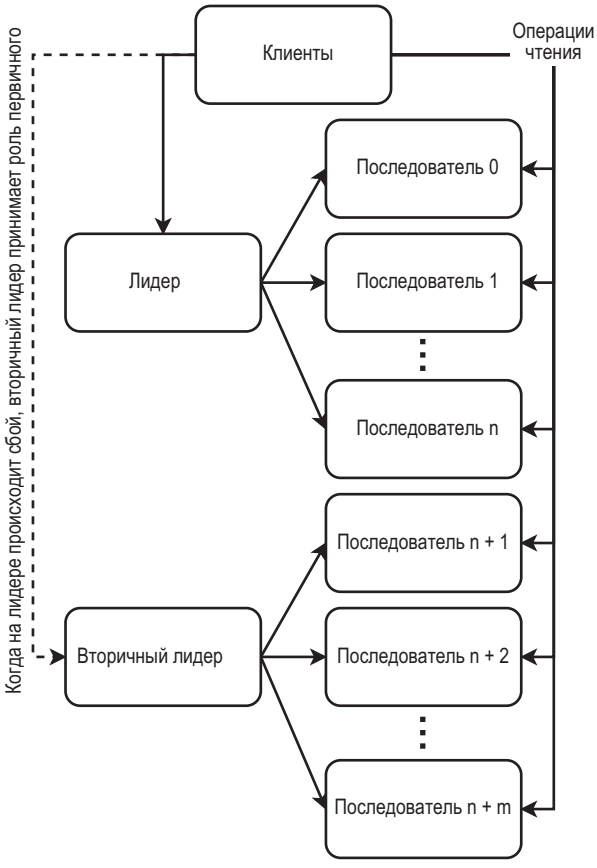

Рис. 4.1. Репликация с одним лидером с отработкой отказа лидера по схеме «первичный-вторичный». Диаграмма взята из книги Артура Эйсмонта (Artur Ejsmont) «Web Scalability for Startup Engineers» (McGraw Hill, 2015)

Один узел имеет максимальную пропускную способность, которая должна совместно использоваться всеми последователями. Тем самым устанавливается максимальное количество последователей, что, в свою очередь, ограничивает масштабируемость чтения. Чтобы продолжить масштабирование, можно использовать многоуровневую репликацию, как показано на рис. 4.2. Последователи образуют несколько уровней в виде пирамиды. Каждый уровень реплицирует данные на следующем нижнем уровне. Каждый узел реплицирует данные на том количестве последователей, какое он способен обработать, но с некоторым ущербом для согласованности.

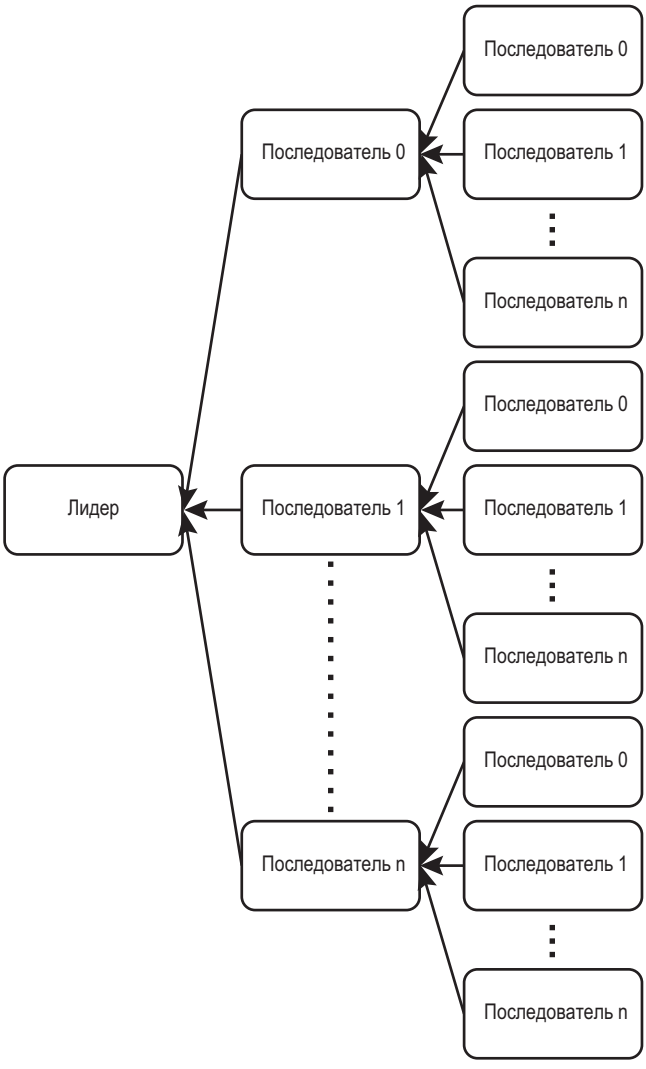

Рис. 4.2. Многоуровневая репликация. Каждый уровень реплицирует данные на своих последователях, которые, в свою очередь, реплицируют данные на своих последователях. Такая архитектура гарантирует, что узел будет выполнять репликацию на том количестве последователей, какое он способен обработать, но с некоторым ущербом для согласованности

Репликация с одним лидером реализуется проще всего. Главное ее ограничение заключается в том, что вся база данных должна размещаться на одном хосте. Другим ограничением становится согласованность в конечном счете, так как репликация записи на последователи требует времени.

Примером репликации с одним лидером служит репликация в MySQL на базе binlog. Она хорошо описана в главе 5 книги Эйсмонта «Web Scalability for Startup Engineers».

Некоторые сетевые ресурсы по теме:

- https://dev.to/tutelaris/introduction-to-mysql-replication-97c
- https://dev.mysql.com/doc/refman/8.0/en/binlog-replication-coniguration-overview.html
- https://www.digitalocean.com/community/tutorials/how-to-set-up-replication-in-mysql
- https://docs.microsoft.com/en-us/azure/mysql/single-server/how-to-data-in-replication
- https://www.percona.com/blog/2013/01/09/how-does-mysql-replication-really-work/
- https://hevodata.com/learn/mysql-binlog-based-replication/

Прием масштабирования репликации с одним лидером: логика запроса в уровне приложения

Вручную вводимые строки медленно увеличивают размер баз данных — в этом можно убедиться при помощи простых оценок и вычислений. Если данные были сгенерированы программным обеспечением или накапливались долгое время, то размер хранимых данных может выйти за пределы одного узла.

Если сократить размер базы данных не удается, но вы хотите продолжать использовать SQL, можно разделить данные между несколькими БД SQL. Это означает, что сервис необходимо будет настроить для подключения к нескольким БД SQL, и запросы SQL в приложении придется переписывать для получения информации из соответствующей базы данных.

Если одну таблицу необходимо распределить между двумя и более базами данных, приложение должно обращаться к разным базам и объединять результаты. Логика запросов уже не инкапсулируется в БД и проникает в приложение. Приложение должно хранить метаданные для отслеживания того, в какой базе данных хранятся те или иные данные. Фактически это репликация с несколькими лидерами и управлением метаданными в приложении. Такой подход усложняет обслуживание сервисов и баз данных, особенно если эти БД используются несколькими сервисами.

Например, если рекомендательная система Beigel обрабатывает миллиарды поисковых запросов ежедневно, таблица SQL `fact_searches`, в которой они сохраняются, вырастет до нескольких терабайт за считаные дни. Эти данные можно распределить между несколькими базами данных, каждая в своем кластере. Можно проводить секционирование по дням, ежедневно создавать новую таблицу и присваивать таблицам имена в формате `fact_searches_ГГГГ_ММ_ДД` (например, `fact_searches_2023_01_01` и `fact_searches_2023_01_02`). Любое приложение, которое запрашивает данные из этих таблиц, должно включать эту логику секционирования, в данном случае схему присваивания имен таблицам. В более

сложном примере некоторые клиенты могут создавать столько транзакций, что для них требуются новые таблицы. Если большое количество запросов к поисковому API отправляется другими рекомендательными приложениями, можно создать таблицу для каждого из них (например, `fact_searches_a_2023_01_01` для хранения всех запросов, поступивших 1 января 2023 года от компаний, названия которых начинаются с буквы A). Также может понадобиться другая таблица SQL `search_orgs`, хранящая метаданные компаний, направляющих поисковые запросы к Beigel.

Такой вариант можно предложить как одну из возможностей, но вы вряд ли будете использовать этот дизайн. При репликации с несколькими лидерами или без лидера следует использовать базы данных.

4.3.3. Репликация с несколькими лидерами

Методы репликации с несколькими лидерами или без лидера предназначены для масштабирования записей и размера баз данных. Они требуют обработки ситуаций гонки, отсутствующих в репликации с одним лидером.

В репликации с несколькими лидерами, как следует из названия, несколько узлов назначаются лидерами, и записи могут выполняться на любом из них. Каждый лидер должен реплицировать свои записи на всех остальных узлах.

Проблемы консистентности и способы решения

Репликация вводит консистентность и ситуацию гонки для операций, в которых важна последовательность. Например, если строка обновляется на одном лидере и удаляется на другом, каким должен быть результат? Упорядочение операций по меткам времени не подходит, потому что синхронизация часов на разных узлах может не быть идеальной. Использовать одни часы на разных узлах не получится, потому что узлы будут получать сигналы часов в разное время — хорошо известное явление, называемое *рассинхронизацией часов*. Таким образом, даже показания часов серверов, которые периодически синхронизируются с одним источником, будут отличаться на несколько миллисекунд и более. Если запросы выдаются к разным сервисам в пределах интервалов времени, меньших этой разности, будет невозможно определить порядок их выполнения.

Здесь мы переходим к проблемам репликации и сценариям, связанным с консистентностью, которые часто обсуждаются на собеседованиях по проектированию систем. Эти ситуации могут возникнуть с любым форматом хранения, включая базы данных и файловые системы. В книге Мартина Клеппмана «Designing Data-Intensive Applications» очень подробно рассматриваются основные ловушки репликации.

Что такое консистентность базы данных? Консистентность гарантирует, что транзакция переводит базу данных из одного действительного состояния

в другое, сохраняя инварианты базы данных; все данные, записанные в базу, должны быть действительными в соответствии со всеми определенными правилами, включая ограничения, каскадные эффекты, триггеры или любые их сочетания.

Как уже говорилось, определение консистентности довольно сложное. Неформально ее понимают как требование, чтобы данные были одинаковыми для всех пользователей:

1. Одинаковые запросы к нескольким репликам должны возвращать одинаковые результаты, несмотря на то что реплики размещаются на разных физических серверах.
2. Запросы DML (например, `INSERT`, `UPDATE` или `DELETE`) к разным физическим серверам, но требующие обработки одних строк данных, должны выполняться в порядке их отправки.

Подходящим вариантом может быть консистентность в конечном счете, но любому пользователю может понадобиться получить данные в актуальном состоянии. Например, если пользователь А запрашивает значение счетчика, увеличивает его на единицу, а затем снова запрашивает значение этого счетчика, для пользователя А будет логично получить значение, увеличенное на единицу. При этом другие пользователи, запрашивающие счетчик, могут получить значение до увеличения. Это правило называется «консистентностью чтения после записи».

В общем случае старайтесь по возможности снижать требования к консистентности. Найдите подходы, минимизирующие объем данных, требующих консистентности для всех пользователей.

Запросы DML к разным физическим серверам, но требующие обработки одних и тех же строк, могут создать ситуацию гонки. Несколько примеров ситуаций:

- Запросы `DELETE` и `INSERT` к одной строке таблицы с первичным ключом. Если первой выполняется операция `DELETE`, то строка должна существовать. Если первой выполняется `INSERT`, то `DELETE` удалит строку.
- Две операции `UPDATE` к одной ячейке с разными значениями. Итоговое состояние должна устанавливать только одна из них.

А что делать с запросами DML, отправленными в одну миллисекунду к разным серверам? Такая ситуация крайне маловероятна, и похоже, общепринятой схемы решения для подобных случаев не существует. Можно предложить несколько решений. Одно из них — выполнить запрос `DELETE` перед `INSERT`/`UPDATE` и случайным образом назначить приоритет запросам `INSERT`/`UPDATE`. В любом случае компетентный эксперт на 50-минутном собеседовании не будет тратить время на обсуждение вопросов, для которых нет определенного решения.

4.3.4. Репликация без лидера

В репликации без лидера все узлы равны. Операции чтения и записи могут выполняться с любым узлом. Как обрабатываются ситуации гонки? Один из способов основан на введении концепции кворума. *Кворум* определяет минимальное количество узлов, которые должны иметь согласованную информацию. Легко сделать вывод, что если база данных имеет n узлов, а для операций чтения и записи установлен кворум $n/2+1$ узлов, то согласованность гарантирована. Если вам нужна согласованность, приходится выбирать между быстрым чтением и быстрой записью. Если нужна быстрая запись, установите низкий кворум для записи и высокий кворум для чтения, и наоборот — для быстрого чтения. В противном случае возможна только согласованность в конечном счете, и операции `UPDATE` и `DELETE` не могут быть согласованными.

Cassandra, Dynamo, Riak и Voldemort — примеры баз данных, использующих репликацию без лидера. В Cassandra операции `UPDATE` подвержены ситуациям гонки, тогда как операции `DELETE` реализуются с использованием tombstone-меток на удаление вместо фактического удаления строк. В HDFS операции чтения и репликации базируются на локальности стоек, и все реплики считаются равными.

4.3.5. Репликация HDFS

Этот раздел поможет вам освежить в памяти основные особенности HDFS, Hadoop и Hive. Подробно обсуждать их мы не будем.

Репликация HDFS не имеет четкого отнесения ни к одной из описанных трех категорий. У кластера HDFS имеется активный узел `NameNode`, пассивный (резервный) узел `NameNode` и несколько узлов `DataNode`. `NameNode` выполняет операции пространства имен файловой системы, такие как открытие, закрытие и переименование файлов и каталогов. Он также определяет отображение блоков на узлы `DataNode`. Узлы `DataNode` отвечают за обслуживание запросов чтения и записи от клиентов файловой системы. Узлы `DataNode` также выполняют операции создания блоков, их удаления и репликации по инструкциям от `NameNode`. Пользовательские данные никогда не переходят через `NameNode`. HDFS сохраняет таблицу в виде одного или нескольких файлов в каталоге. Каждый файл делится на блоки, шардируемые между узлами `DataNode`. Размер блока по умолчанию равен 64 Мбайт; это значение может задаваться администраторами.

Hadoop — фреймворк для хранения и обработки распределенных данных с использованием модели программирования MapReduce. Hive — технология хранилища данных, построенная на базе Hadoop. В Hive существует концепция секционирования таблиц по одному или нескольким столбцам для эффективных запросов фильтрации. Например, секционированная таблица Hive создается следующим образом:

```
CREATE TABLE sample_table (user_id STRING, created_date DATE,
country STRING) PARTITIONED BY (created_date, country);
```

На рис. 4.3 изображено дерево каталогов этой таблицы. Каталог таблицы содержит подкаталоги для значений даты, в которых, в свою очередь, содержатся подкаталоги для значений столбцов. При фильтрации запросов по `created_date` и/или `country` будут обработаны только соответствующие файлы, что позволит избежать затрат на полное сканирование таблицы.

Рис. 4.3. Пример дерева каталогов HDFS для таблицы sample_table со столбцами даты (created_date) и страны (country); таблица секционируется по двум столбцам. Каталог sample_table содержит подкаталоги для значений created_date, которые, в свою очередь, содержат подкаталоги для значений country. (Источник: https://stackoverlow.com/questions/44782173/hive-does-hive-support-partitioning-and-bucketing-while-usiing-external-tables)

HDFS поддерживает только добавление данных и не поддерживает операции `UPDATE` или `DELETE` — возможно, из-за риска состояний гонки при репликации с `UPDATE` и `DELETE`. Для операции `INSERT` состояний гонки нет.

В HDFS поддерживаются квоты имен, пространственные квоты и квоты типов хранения данных. Для дерева каталогов:

- Квота имен — жесткий лимит количества имен файлов и каталогов.
- Пространственная квота — жесткий лимит количества байтов во всех файлах.
- Квота типов хранения данных — жесткий лимит на использование конкретных типов хранения данных. Обсуждение типов хранения данных в HDFS выходит за рамки книги.

СОВЕТ Начинающие пользователи Hadoop и HDFS часто используют команду Hadoop INSERT, чего делать не рекомендуется. Запрос INSERT создает новый файл с одной строкой, который занимает полный блок из 64 Мбайт, что весьма неэффективно. Кроме того, он получает имя, и программные запросы INSERT быстро превысят квоту имен. За дополнительной информацией обращайтесь к https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsQuotaAdminGuide.html. Используйте прямое присоединение к файлу HDFS и при этом следите, чтобы присоединяемые строки содержали те же поля, что и существующие строки в файле, чтобы избежать нарушений целостности данных и ошибок обработки.

Если вы работаете с фреймворком Spark, хранящим данные в HDFS, используйте `saveAsTable` или `saveAsTextFile`, как в следующем фрагменте кода. За подробностями обращайтесь к документации Spark https://spark.apache.org/docs/latest/sql-data-sources-hive-tables.html.

```
val spark = SparkSession.builder().appName("Our app").config("some.config",
     "value").getOrCreate()
val df = spark.sparkContext.textFile({hdfs_file})
df.createOrReplaceTempView({table_name})
spark.sql({spark_sql_query_with_table_name}).saveAsTextFile({hdfs_directory})
```

4.3.6. Дополнительная литература

Следующие темы более подробно рассматриваются в книге Мартина Клеппмана «Designing Data-Intensive Applications»[1]:

- Техники согласованности: разрешение конфликтов при чтении, антиэнтропия и кортежи.
- Консенсусный алгоритм репликации с несколькими лидерами и реализации в CouchDB, репликации MySQL Group и Postgres.
- Проблемы отработки отказов (например, «расщепление сознания»).
- Разные консенсусные алгоритмы для разрешения ситуаций гонки. Консенсусный алгоритм предназначен для достижения соглашения о значении данных.

4.4. МАСШТАБИРОВАНИЕ ЕМКОСТИ ХРАНИЛИЩА С ШАРДИРОВАННЫМИ БАЗАМИ ДАННЫХ

Если размер базы данных превышает емкость одного хоста, старые записи нужно будет удалить. Если вам нужны старые данные, храните их в шардированном хранилище, таком как HDFS или Cassandra. Шардированное хранилище является горизонтально масштабируемым и теоретически должно поддерживать бесконечную емкость за счет простого добавления новых хостов. Существуют рабочие кластеры HDFS емкостью более 100 Пбайт (https://eng.uber.com/uber-big-data-platform/). Теоретически возможны кластеры емкостью, измеряемой в йоттабайтах, но финансовые затраты на оборудование, необходимое для хранения и анализа таких объемов данных, будут неприемлемо высокими.

> **СОВЕТ** Для хранения данных, используемых для прямого обслуживания потребителей, можно использовать базу данных с низкой задержкой, такую как Redis.

[1] Клеппман М. «Высоконагруженные приложения. Программирование, масштабирование, поддержка». СПб., издательство «Питер».

Другой вариант — хранение данных на устройствах потребителей или в браузерных cookie и localStorage. Однако в таком случае все операции обработки данных должны выполняться на фронтенде, а не на бэкенде.

4.4.1. Шардированные РСУБД

Если вам нужно использовать РСУБД и объем данных превышает тот, который может храниться на одном узле, можно воспользоваться шардированной РСУБД, такой как Amazon RDS (https://aws.amazon.com/blogs/database/sharding-with-amazon-relational-database-service/), или реализовать собственное шардированное решение. Все они подразумевают ряд ограничений для операций SQL:

- Запросы `JOIN` будут работать намного медленнее. Они будут требовать большого объема сетевого трафика при передаче данных от одного узла ко всем остальным узлам. Представим запрос `JOIN` между двумя таблицами по конкретному столбцу. Если обе таблицы шардируются между узлами, каждый шард одной таблицы должен сравнить значение этого столбца в каждой строке с тем же столбцом в каждой строке другой таблицы. Намного эффективнее, если операция `JOIN` будет выполняться со столбцами, используемыми как ключи шардов, так как каждый узел будет знать, с какими другими узлами необходимо соединение. Операции `JOIN` можно ограничить только такими столбцами.
- Операции агрегирования будут включать базу данных и приложение. Одни операции агрегирования проще других (например, вычисление суммы или среднего). Каждый узел просто суммирует и/или подсчитывает значения, а затем возвращает агрегированные значения приложению, которое выполняет простые арифметические вычисления для получения окончательного результата. Другие операции агрегирования (такие, как медиана или процентили) сложнее и медленнее.

4.5. АГРЕГИРОВАНИЕ СОБЫТИЙ

Операции записи в базы данных сложны, а их масштабирование обходится дорого, поэтому лучше сокращать частоту записи в базы данных там, где это возможно, на уровне дизайна системы. Выборка и агрегирование — стандартные методы сокращения частоты записи в базы данных. Дополнительное их преимущество — замедление роста БД.

Кроме сокращения частоты записи в базы данных, можно снизить частоту чтения из базы данных; для этого используются такие методы, как кэширование и аппроксимация. В главе 17 рассматривается Count-Min Sketch — алгоритм создания таблицы приблизительных частот для событий в непрерывных потоках данных.

Термин «*выборка*» (sampling) означает, что используются только некоторые точки данных, а другие игнорируются. Существует много стратегий выборки, такие как выборка каждой *n*-й точки данных или случайная выборка. Выборка сама по себе довольно проста, и ее стоит упомянуть во время собеседования.

Агрегирование событий подразумевает объединение / комбинирование событий в одном событии, так что вместо многих операций записи в базу данных выполняется только одна операция. Оно подходит, если точные метки времени отдельных событий не важны.

Агрегирование реализуется с использованием потокового пайплайна. Первая стадия потокового пайплайна может получать события с высокой частотой и требует большого кластера с тысячами хостов. Без агрегирования каждая последующая стадия также потребует большого кластера. Благодаря агрегированию каждая последующая стадия может не иметь хостов. Также применяется репликация и контрольные точки на случай сбоя хостов. Как показано в главе 5, мы можем использовать алгоритм распределенных транзакций (такой, как сага) или кворум для записи, чтобы количество реплик для каждого события было минимальным.

4.5.1. Одноуровневое агрегирование

Агрегирование может быть одноуровневым или многоуровневым. На рис. 4.4 изображен пример одноуровневого агрегирования при подсчете количества значений. В этом примере событие может иметь значение A, B, C и т. д.

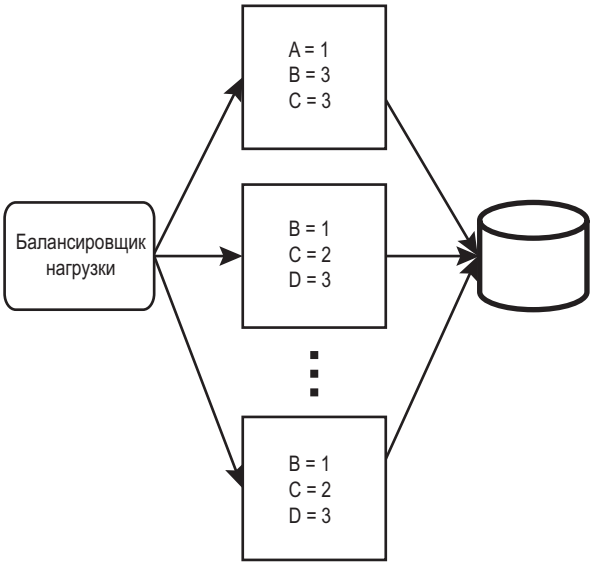

Рис. 4.4. Пример одноуровневого агрегирования. Балансировщик нагрузки распределяет события по одному уровню хостов, который агрегирует их, а затем записывает агрегированные значения в базу данных. Если бы отдельные события записывались прямо в базу данных, частота записи была бы заметно выше и базу данных пришлось бы масштабировать. На диаграмме не показаны реплики хостов, обязательные при необходимости высокой доступности и точности

Эти события могут равномерно распределяться по хостам балансировщиком нагрузки. Каждый хост хранит хеш-таблицу в памяти и агрегирует эти счетчики в своей хеш-таблице. Каждый хост может записывать счетчики в базу данных через определенный интервал времени (например, каждые 5 минут) или при нехватке памяти (в зависимости от того, что произойдет быстрее).

4.5.2. Многоуровневое агрегирование

На рис. 4.5 изображено многоуровневое агрегирование. Каждый уровень хостов может агрегировать события своих предков на предыдущем уровне. Количество хостов на каждом уровне можно последовательно сокращать, пока не будет достигнуто нужное количество хостов (оно определяется требованиями и доступными ресурсами) на последнем уровне, на котором осуществляется запись в базу данных.

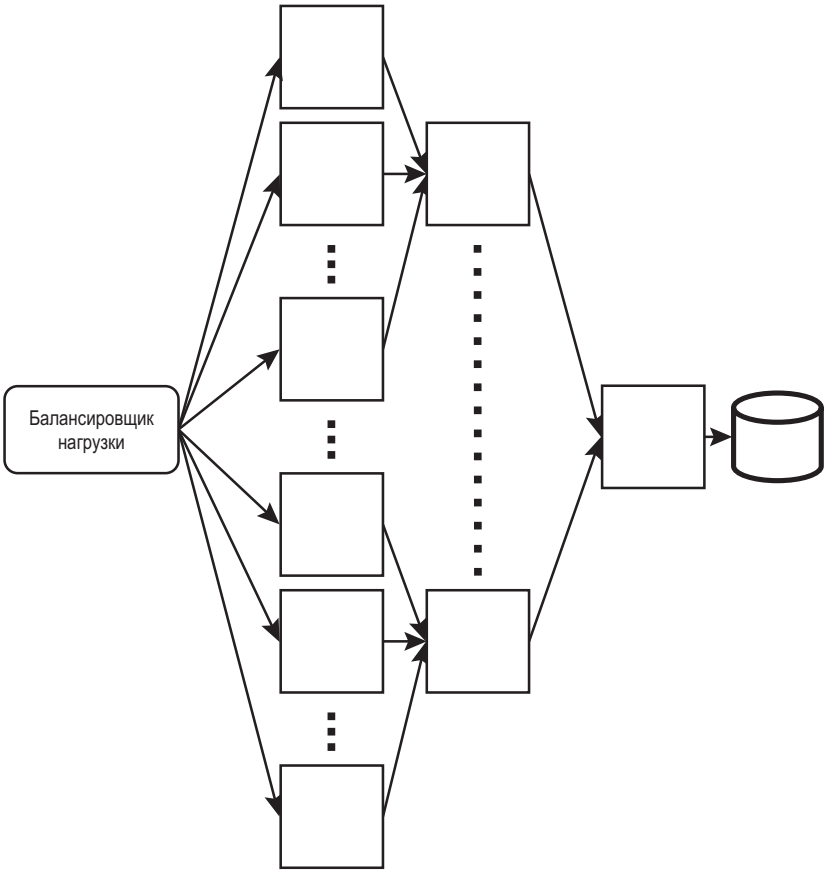

Рис. 4.5. Структура многоуровневого агрегирования напоминает инвертированную структуру многоуровневой репликации

Главные минусы агрегирования — согласованность в конечном счете и высокая сложность. Каждый уровень добавляет задержку в пайплайн и как следствие — в операции записи в базу данных, так что результаты чтения из базы данных могут оказаться устаревшими. Реализация репликации, записи в журнал, мониторинга и оповещений тоже может добавить сложности в эту систему.

4.5.3. Секционирование

Этот метод требует балансировщика нагрузки 7-го уровня. (Краткое описание балансировщика нагрузки 7-го уровня см. в разделе 3.1.2.) Балансировщик нагрузки можно настроить для обработки входящих запросов и перенаправления их определенным хостам в зависимости от содержимого событий.

Рассмотрим пример на рис. 4.6. Если события содержат простые значения A–Z, балансировщик нагрузки можно настроить для перенаправления событий со значениями A–I одним хостам, событий со значениями J–R — другим хостам и событий со значениями S–Z — третьим. Хеш-таблицы с 1-го уровня хостов агрегируются на 2-м уровне хостов, а затем на хосте последней хеш-таблицы. Наконец, эта хеш-таблица отправляется хосту max-heap[1], который строит итоговую таблицу.

Можно ожидать, что трафик событий будет подчиняться нормальному распределению, а следовательно, некоторые партиции будут получать непропорционально высокий трафик. Из рис. 4.6 видно, что каждой партиции можно выделить разное количество хостов. В партицию A–I входят три хоста, в партицию J–R — один хост, а в S–Z — два хоста. Мы распределили узлы так, поскольку трафик между партициями неравномерен, а некоторые хосты могут получать непропорционально высокий трафик (то есть становятся «горячими») — больший, чем они способны обработать.

Также заметим, что партиция J–R содержит только один хост, поэтому у нее нет 2-го уровня. Нам как проектировщикам приходится принимать такие решения в зависимости от ситуации.

Кроме выделения разного количества хостов в каждую партицию, другой способ равномерного распределения трафика основан на регулировке количества и ширины партиций. Например, вместо {A-I, J-R, S-Z} можно создать партиции {{A-B, D-F}, {C, G-J}, {K-S}, {T-Z}}. Другими словами, мы перешли от трех партиций к четырем и включили C во вторую партицию. Для удовлетворения требований к масштабируемости системы можно действовать креативно и менять подходы.

---

[1] max-heap — двоичная куча, в которой значение в любой вершине не меньше, чем значения ее потомков. — *Примеч. ред.*

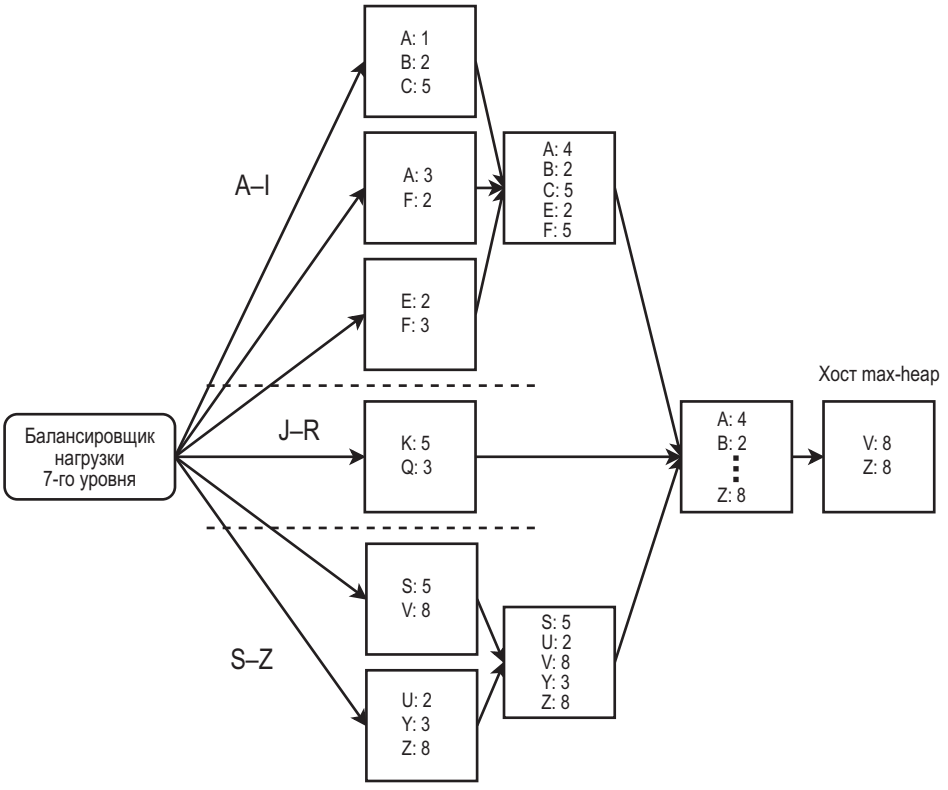

Рис. 4.6. Многоуровневое агрегирование с секционированием

4.5.4. Большие пространства ключей

На рис. 4.6 в предыдущем разделе изображено крошечное пространство из 26 ключей от A до Z. В практической реализации пространство ключей будет намного больше. Необходимо добиться того, чтобы объединенные пространства ключей конкретного уровня не вызвали переполнения памяти на следующем уровне. Хосты предшествующих уровней агрегирования должны ограничивать свое пространство имен ключей объемом, который может поместиться в их памяти, чтобы хосты последующих уровней агрегирования имели достаточно памяти для всех ключей. Это может означать, что хостам более ранних уровней агрегирования придется чаще записывать свои данные.

Например, на рис. 4.7 изображен простой сервис агрегирования всего с двумя уровнями. На первом уровне два хоста, а на втором — один. Два хоста первого уровня должны ограничить свое пространство ключей половиной от фактического, чтобы хост второго уровня мог вместить все ключи.

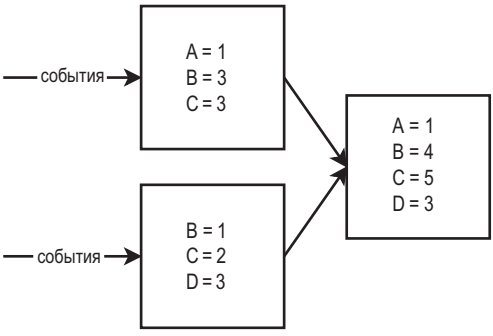

Рис. 4.7. Простой сервис агрегирования с двумя уровнями: два хоста на первом уровне, один хост на втором уровне

Хосты первых уровней агрегирования также могут иметь меньший объем памяти, а хосты последних уровней — больший.

4.5.5. Репликация и отказоустойчивость

Мы еще не обсуждали репликацию и отказоустойчивость. Если хост выйдет из строя, он потеряет все агрегированные события. Более того, сбой будет каскадным, потому что на всех предшествующих хостах может произойти переполнение, и агрегированные события тоже будут потеряны.

Для решения проблемы можно воспользоваться контрольными точками или очередями недоставленных сообщений, описанными в разделах 3.3.6 и 3.3.7. Но поскольку отказ хоста на глубоком уровне может повлиять на работу множества хостов, придется повторить значительный объем вычислений, а это неэффективное расходование ресурсов. Такой сбой может значительно увеличить задержку агрегирования.

Одно из возможных решений — преобразование каждого узла в независимый сервис при помощи кластера из нескольких узлов без сохранения состояния, которые направляют запросы к общей базе данных в памяти (например, Redis). Такой сервис изображен на рис. 4.8. Сервис может иметь несколько хостов (например, три хоста без сохранения состояния). Общий сервис балансировки нагрузки может распределять запросы между этими хостами. Масштабирование проблемой не является, так что каждый сервис может ограничиться небольшим количеством хостов (например, тремя для отказоустойчивости).

В начале главы мы отмечали нежелательность лишних операций записи в базы данных, что на первый взгляд противоречит сказанному здесь. Однако каждый сервис располагает отдельным кластером Redis, поэтому конкуренции за запись с одним ключом не будет. Более того, агрегированные события удаляются при каждой успешной записи, так что база данных не будет расти неконтролируемо.

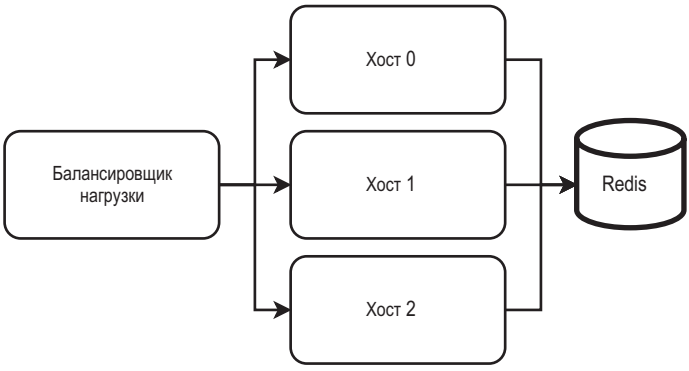

Рис. 4.8. Узел можно заместить сервисом, называемым блоком агрегирования. Блок содержит три хоста без сохранения состояния для обеспечения отказоустойчивости, но при желании можно использовать больше хостов

> **ПРИМЕЧАНИЕ** Полностью сервис агрегирования можно организовать в Terraform. Каждый блок агрегирования может представлять собой кластер Kubernetes с тремя модулями и одним хостом на модуль (или двумя хостами, если вы используете паттерн sidecar).

4.6. ПАКЕТНЫЕ И ПОТОКОВЫЕ ETL

ETL (Extract, Transform, Load, то есть «извлечение, преобразование, загрузка») — общая процедура копирования данных из одного или нескольких источников в систему-получатель, в которой данные представляются иначе, чем в источнике (источниках), или в ином контексте, чем в источнике (источниках). Под «пакетным режимом» понимается обработка данных группами (пакетами) — обычно это делается периодически, но обработка может инициироваться и вручную. Потоковым режимом называется непрерывный поток данных, которые должны обрабатываться в реальном времени.

Можно провести аналогию между пакетным/потоковым режимом и режимами опроса/прерывания. Как и опрос, пакетное задание всегда выполняется с определенной частотой независимо от того, имеются ли новые события для обработки, тогда как потоковое задание запускается каждый раз, когда выполняется условие-триггер (обычно это публикация нового события).

Пример сценария, в котором уместно применять пакетные задания, — генерация ежемесячных счетов (обычно в форматах PDF или CSV) для клиентов. Подобные пакетные задания особенно актуальны, если данные, необходимые для выставления счетов, становятся доступными только в определенный день каждого месяца (например, когда поставщик предоставляет данные тарификации, необходимые для генерации счетов). Если все данные для создания этих периодических файлов генерируются внутри организации, можно рассмотреть

архитектуру Kappa (см. главу 17) и реализовать потоковое задание, которое обрабатывает каждый блок данных сразу же после его появления. Преимущества такого подхода — появление доступных файлов практически сразу же после завершения месяца, распределение затрат на обработку данных на все дни месяца и простота отладки функции, обрабатывающей небольшие блоки данных, по сравнению с пакетным заданием, которое обрабатывает гигабайты данных.

В категории пакетных инструментов популярны Airflow и Luigi, а в категории потоковых инструментов — Kafka и Flink. Flume и Scribe — специализированные потоковые инструменты для ведения журналов; они накапливают данные журналов со многих серверов в реальном времени. Ниже кратко представлены некоторые концепции ETL.

Пайплайн ETL состоит из направленного ациклического графа (НАГ) задач. В НАГ узел соответствует задаче, а его предки — его зависимостям. Задание соответствует одному проходу пайплайна ETL.

4.6.1. Простой пакетный пайплайн ETL

Простой пакетный пайплайн ETL может быть реализован с использованием crontab, двух таблиц SQL и скрипта (программы, написанной на скриптовом языке) для каждого задания. cron подойдет для небольших и не имеющих критической важности задач без параллелизма, для которых достаточно одной машины. Два примера таблиц SQL:

```
CREATE TABLE cron_dag (
  id INT,         -- идентификатор задачи.
  parent_id INT,  -- Родительская задача. Задача может иметь 0, 1 или
                  --несколько родителей.
  PRIMARY KEY (id),
  FOREIGN KEY (parent_id) REFERENCES cron_dag (id)
);
CREATE TABLE cron_jobs (
  id INT,
  name VARCHAR(255),
  updated_at INT,
  PRIMARY KEY (id)
);
```

В инструкциях crontab может содержаться список скриптов. В данном примере используются скрипты Python, хотя подойдет любой скриптовый язык. Все скрипты можно разместить в общем каталоге /cron_dag/dag/, а другие файлы/модули Python — в других каталогах.

Правил, определяющих способ организации файлов, не существует; на наш взгляд, оптимальна следующая схема:

```
0 * * * * ~/cron_dag/dag/first_node.py
0 * * * * ~/cron_dag/dag/second_node.py
```

Каждый скрипт может действовать по следующему алгоритму. Шаги 1 и 2 абстрагируются в модули, пригодные для повторного использования:

1. Проверить, что значение `updated_at` соответствующей задачи меньше, чем у его зависимых задач.
2. Запустить мониторинг при необходимости.
3. Выполнить указанную задачу.

Главные недостатки такого подхода:

- Отсутствие масштабируемости. Все задачи выполняются на одном хосте, что подразумевает типичные недостатки решений с одним хостом:
  - наличие единой точки отказа;
  - риск нехватки вычислительных ресурсов для запуска всех задач, запланированных на конкретное время;
  - риск превышения емкости хранилища на хосте.
- Задача может включать в себя множество меньших задач: например, рассылки оповещений по миллионам устройств. Если при выполнении такой задачи произойдет отказ, необходимо избежать повторения малых задач, завершившихся успехом (то есть отдельные задачи должны быть идемпотентными). Рассмотренный простой дизайн не обеспечивает такой идемпотентности.
- Отсутствие инструментов проверки согласованности идентификаторов задач в скриптах Python и таблицах SQL. Таким образом, эта структура подвержена ошибкам программирования.
- Отсутствие графического интерфейса (если только вы не создадите его самостоятельно).
- Мы еще не реализовали ведение журналов, мониторинг и оповещения. Это очень важно, и этим следует заняться на следующем шаге. Например, что, если задача завершится отказом или на хосте произойдет сбой при ее выполнении? Необходимо, чтобы запланированные задачи завершились успешно.

ВОПРОС Как провести горизонтальное масштабирование простого пакетного пайплайна ETL для улучшения его масштабируемости и доступности?

К числу специализированных систем планирования задач принадлежат Airflow и Luigi. Эти средства включают веб-интерфейсы для визуализации НАГ и удобства взаимодействия. Они также являются вертикально масштабируемыми и могут выполняться в кластерах, управляя большим количеством задач. В этой книге мы будем использовать общий сервис Airflow уровня организации, если нам понадобится пакетный сервис ETL.

4.6.2. Терминология, касающаяся передачи сообщений

В этом разделе объясняются термины, принятые для разных типов конфигураций передачи сообщений и потоковой передачи, часто встречающиеся в технических обсуждениях или литературе.

Система передачи сообщений (messaging system)

Это общий термин для обозначения системы, передающей данные от одного приложения к другому. Ее цель — уменьшить сложности в передаче и в совместном использования данных в приложениях, чтобы разработчики могли сосредоточиться на обработке данных.

Очередь сообщений (message queue)

Сообщение содержит рабочий объект инструкций, передаваемых от одного сервиса другому, ожидающему обработки в очереди. Каждое сообщение обрабатывается одним потребителем только один раз.

Производитель/потребитель (producer/consumer)

«Производитель/потребитель» (или «издатель/подписчик» (pub/sub)) — асинхронная система передачи сообщений, отделяющая сервисы, производящие события, от сервисов, обрабатывающих события. Система «производитель/потребитель» содержит одну или несколько очередей сообщений.

Брокер сообщений (message broker)

Это программа, преобразующая сообщение из формального протокола передачи сообщений, используемого отправителем, к формальному протоколу передачи сообщений получателя. Брокер сообщений представляет собой уровень преобразования. И Kafka, и RabbitMQ являются брокерами сообщений. RabbitMQ позиционируется как «самый распространенный брокер сообщений с открытым исходным кодом» (https://www.rabbitmq.com/). AMQP — один из протоколов передачи сообщений, реализованный RabbitMQ (описание AMQP выходит за рамки этой книги). Kafka реализует собственный нестандартный протокол передачи сообщений.

Потоковая передача событий (event streaming)

Это общий термин для обозначения непрерывного потока событий, обрабатываемых в реальном времени. Событие содержит информацию об изменении состояния. Kafka — самая распространенная платформа потоковой передачи событий.

push/pull-модели коммуникации

Межпроцессные коммуникации бывают двух видов. В общем случае коммуникация, инициированная потребителем (pull), предпочтительнее коммуникации, инициированной производителем (push), — это одна из основных концепций,

заложенных в основу архитектур «производитель/потребитель». В модели pull потребитель управляет частотой потребления сообщений и предотвращает перегрузку.

В ходе разработки потребителя могут выполняться нагрузочные и стресс-тесты с последующим мониторингом его пропускной способности и производительности на реальном трафике и сравнением результатов с тестовыми. Это позволит команде точно определить, потребуется ли больше инженерных ресурсов для улучшения тестов. Потребитель может отслеживать свою пропускную способность и размеры очередей производителя во времени, а команда может масштабировать его по мере надобности.

Если рабочая система непрерывно находится под высокой нагрузкой, очередь вряд ли будет оставаться пустой в течение сколько-нибудь значительного времени, а потребитель будет продолжать запрашивать сообщения. Если возникнет ситуация, в которой приходится поддерживать большой потоковый кластер для обработки непрогнозируемых выбросов трафика в пределах нескольких минут, этот кластер также следует использовать для других низкоприоритетных сообщений (например, стандартных сервисов Flink, Kafka или Spark для организации).

Другая ситуация, в которой опрос или запрос на отправку данных производителем лучше коммуникации, инициированной производителем, — когда пользователь находится за брандмауэром или зависимость часто изменяется и выдает слишком много push-запросов. Процесс настройки pull-коммуникации также включает на один этап меньше, чем для push-коммуникации, — пользователь уже направляет запросы к зависимости. При этом зависимость обычно не отправляет запросы к пользователю.

Недостаток pull-модели (https://engineering.linkedin.com/blog/2019/data-hub) заключается в том, что если система собирает данные из многих источников с использованием ботов, разработка и обслуживание этих ботов могут быть слишком сложными и трудозатратными. Возможно, решение, в котором отдельные провайдеры данных инициируют отправку данных в центральный репозиторий, будет лучше масштабироваться. Push-модель также позволяет быстрее получать обновления.

Еще один сценарий, в которой push-модель предпочтительнее, встречается в таких приложениях с потерей данных, как сервисы аудио- и видеостриминга. Эти приложения не отправляют повторно данные, которые не были доставлены с первой попытки, и обычно используют UDP для отправки данных получателям.

4.6.3. Kafka и RabbitMQ

На практике в большинстве компаний имеется общий сервис Kafka, используемый другими сервисами. В оставшейся части книги мы будем использовать Kafka везде, где нам понадобится сервис передачи сообщений или потоковой

передачи событий. Чтобы не раздражать излишне категоричных экспертов, безопаснее продемонстрировать знание подробностей и Kafka, и RabbitMQ, различий между ними, а также их плюсов и минусов.

Оба решения могут использоваться для сглаживания неравномерного трафика, чтобы сервис не перегружался выбросами трафика и оставался экономически эффективным: не придется выделять большое количество хостов просто для обслуживания периодов повышенного трафика.

Kafka сложнее RabbitMQ и предоставляет расширенный набор функциональности по сравнению с RabbitMQ. Другими словами, Kafka всегда может использоваться вместо RabbitMQ, но не наоборот.

Если для системы достаточно возможностей сервиса RabbitMQ, мы рекомендуем использовать именно его. Но если в организации уже поддерживается сервис Kafka, лучше остаться на нем, чтобы избежать лишней работы по настройке и обслуживанию (включая ведение журналов, мониторинг и оповещения) другого инструмента, например RabbitMQ. В табл. 4.1 перечислены основные различия между Kafka и RabbitMQ[1].

Таблица 4.1. Некоторые различия между Kafka и RabbitMQ

Kafka	RabbitMQ
Основные цели — масштабируемость, надежность и доступность. Требует более сложной настройки, чем RabbitMQ	Легко настраивается, но не масштабируется по умолчанию
Требует ZooKeeper для управления кластером Kafka[2]. Для этого необходимо настроить IP-адреса всех хостов Kafka в ZooKeeper	Масштабируемость можно самостоятельно реализовать на уровне приложения, присоединив приложение к балансировщику нагрузки и производя/потребляя события от балансировщика нагрузки. Однако это потребует бóльших усилий по настройке, чем с Kafka, а значительно менее зрелая технология будет уступать ей во многих отношениях
Брокер сообщений обладает высокой надежностью благодаря репликации. Можно настроить коэффициент репликации в ZooKeeper и организовать выполнение репликации на разных серверных стойках и датацентрах	Не масштабируется и по умолчанию не обеспечивает надежность. Сообщения теряются при возникновении простоев. Поддерживает функциональность «отложенной очереди» с сохранением сообщений на диске для повышения надежности, но эта функциональность не защищает от отказов дисков на хосте

[1] Рассматривается версия RabbitMQ 3.9 от 26 июля 2021 г. — *Примеч. ред.*

[2] На момент выхода книги в текущей версии Kafka 3.7 Zookeeper уже не является обязательным (см. https://kafka.apache.org/downloads) и планируется к удалению в версии 4.0. — *Примеч. ред.*

Kafka	RabbitMQ
События в очереди не удаляются после потребления, так что одно событие может потребляться многократно. Это необходимо для обеспечения отказоустойчивости на случай отказа потребителя до того, как он завершит обработку события и ему потребуется обработать событие повторно	Сообщения в очереди удаляются при выведении из очереди, как следует из определения «очереди» (RabbitMQ 3.9, версия от 26 июля 2021 года, поддерживает функциональность потоков данных https://www.rabbitmq.com/streams.html, которая позволяет повторно потреблять все сообщения, так что это различие встречается только в старых версиях)
В этом отношении концептуально неточно использовать термин «очередь» в Kafka. На самом деле это список. Тем не менее термин «очередь Kafka» часто встречается на практике	Можно создать несколько очередей, чтобы у сообщения могло быть несколько потребителей (по одной очереди на потребителя). Тем не менее множественные очереди создаются не для этого
В Kafka можно настроить период удержания данных (по умолчанию 7 дней), так что событие будет удалено через 7 дней независимо от того, было оно потреблено или нет. Можно выбрать бесконечный период удержания и использовать Kafka как базу данных	Поддерживает концепцию AMQP для стандартных приоритетов очередей на уровне сообщений. Можно создать несколько очередей с разными приоритетами. Сообщения очереди не извлекаются из нее до тех пор, пока не будут очищены все очереди с более высокими приоритетами. Не существует концепции справедливого распределения, фактор истощения ресурсов также не учитывается
Отсутствие концепции приоритетов	

4.6.4. Лямбда-архитектура

Лямбда-архитектура представляет собой архитектуру обработки больших объемов данных, с параллельным выполнением пакетных и потоковых пайплайнов. По-простому это означает наличие параллельных быстрых и медленных пайплайнов, обновляющих один и тот же приемник. Быстрый пайплайн жертвует согласованностью и точностью для снижения задержки (быстрые обновления), медленный — наоборот. Быстрые пайплайны применяют такие средства, как:

- алгоритмы аппроксимации (см. раздел 17.7);
- базы данных в памяти (например, Redis);
- для ускорения обработки узлы быстрого пайплайна могут не выполнять репликацию данных, которые они обрабатывают, что может привести к потере данных и снижению точности при отказах узлов.

Медленный пайплайн обычно использует базы данных MapReduce, такие как Hive и Spark с HDFS. Мы можем рекомендовать лямбда-архитектуру для систем, использующих большие данные и требующих согласованности и точности.

О разных решениях для баз данных

Существует множество решений для баз данных. Наиболее популярные из них — всевозможные дистрибутивы SQL, Hadoop и HDFS, Kafka, Redis и Elasticsearch. Также имеется большое количество менее распространенных решений, включая MongoDB, Neo4j, AWS DynamoDB и Google Firebase Realtime Database. Как правило, на собеседовании по проектированию систем от вас не ожидают знания таких технологий, особенно проприетарных. Проприетарные базы данных используются редко. Если стартап принимает проприетарную БД, ему стоит как можно скорее перейти на БД с открытым исходным кодом. Чем больше база данных, тем больше проблем создает привязка к конкретной компании-разработчику, так как процесс миграции становится более сложным, рискованным и затратным.

У лямбда-архитектуры есть альтернатива: каппа-архитектура. *Каппа-архитектура* представляет собой паттерн программных архитектур для обработки потоковых данных, когда как пакетная, так и потоковая обработка выполняется в одном технологическом стеке. Она использует для хранения входных данных журнал только для записи (как в Kafka), с последующей потоковой обработкой и хранением в базе данных, к которой пользователи обращаются с запросами. Подробное сравнение лямбда- и каппа-архитектур представлено в разделе 17.9.1.

4.7. ДЕНОРМАЛИЗАЦИЯ

Если данные сервиса помещаются на одном хосте, стандратное решение — выбрать SQL и нормализовать схему. Нормализация имеет ряд преимуществ:

- Данные согласованы, не содержат дубликатов, и в таблицах исключаются нарушения целостности данных.
- Вставки и обновления выполняются быстрее, так как обращаться с запросами нужно только к одной таблице. В денормализованной схеме при операциях вставки или удаления иногда приходится запрашивать данные из нескольких таблиц.
- Уменьшение размера базы данных из-за отсутствия дубликатов. В меньших по объему таблицах операции чтения выполняются быстрее.
- Нормализованные таблицы обычно содержат меньше столбцов; следовательно, у них меньше индексов. Индексы будут перестраиваться быстрее.
- Запросы могут соединять (`JOIN`) только необходимые таблицы.

У нормализации есть и недостатки:

- Запросы с `JOIN` выполняются намного медленнее, чем запросы к отдельным таблицам. На практике денормализация часто выполняется именно по этой причине.

- Таблицы содержат коды, а не данные, так что большинство запросов как для сервиса, так и для ситуативной аналитики будет содержать операции `JOIN`. Запросы `JOIN` обычно более подробные, чем запросы к отдельным таблицам, поэтому их сложнее писать и обслуживать.

На собеседованиях часто заходит речь о таком способе ускорения операций чтения, как компромисс между занимаемым пространством и скоростью за счет денормализации схемы, чтобы в ней не использовались запросы `JOIN`.

4.8. КЭШИРОВАНИЕ

Для баз данных, хранящих данные на диске, частые или недавние запросы могут кэшироваться в памяти. В организациях пользователям могут предоставляться различные технологии баз данных в виде общих сервисов, например сервиса SQL или Spark с HDFS. Эти сервисы также могут применять кэширование, например, с кэшем Redis.

В этом разделе кратко описываются разные стратегии кэширования. Кэширование помогает улучшить следующие параметры системы:

- *Производительность* — основное преимущество; другие преимущества, перечисленные ниже, вторичны. Кэш использует память, которая работает быстрее, но обходится дороже, чем база данных, использующая диски.
- *Доступность* — даже если база данных недоступна, сервис остается доступным, а приложения могут получать данные из кэша (конечно, это относится только к кэшированным данным). Для экономии затрат в кэше может находиться только небольшое подмножество данных из базы. Однако кэши проектируются прежде всего для высокой производительностя и низкой задержки, а не для высокой доступности. Кэш может жертвовать доступностью и другими нефункциональными требованиями ради высокой производительности. База данных должна иметь высокую доступность, и доступность сервиса не должна зависеть от кэша.
- *Масштабируемость* — предоставляя часто запрашиваемые данные, кэш может взять на себя большую часть нагрузки сервиса. Он работает быстрее базы данных, а значит, запросы будут обслуживаться быстрее, что снижает количество открытых в любой момент времени подключений HTTP, и кластер бэкенда меньшего размера может справляться с той же нагрузкой. Тем не менее применять этот метод масштабирования не рекомендуется, если кэш проектируется для оптимизации задержки и для достижения этой цели может жертвовать доступностью. Например, если кэш не реплицируется между датацентрами, поскольку запросы между датацентрами выполняются медленно, что противоречит основной цели кэша — снижению задержки. Таким образом, если в датацентре произойдет сбой (например, из-за проблем с сетью), кэш станет недоступным и вся нагрузка перейдет на базу данных,

которая может с ней не справиться. Бэкенд-сервис должен использовать ограничение частоты, отрегулированное по емкости бэкенда и базы данных.

Кэширование может выполняться на многих уровнях, включая уровни клиента, шлюза API (см. Rob Vettor, David Coulter, Genevieve Warren, Caching in a cloud-native application, Microsoft Docs. 17 мая 2020 г.) (https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/azure-caching) и каждого сервиса (см. Cornelia Davis, Cloud Native Patterns (Manning Publications, 2019)). На рис. 4.9 представлена схема кэширования на уровне шлюза API. Кэш может масштабироваться независимо от сервисов для обслуживания трафика в любой момент времени.

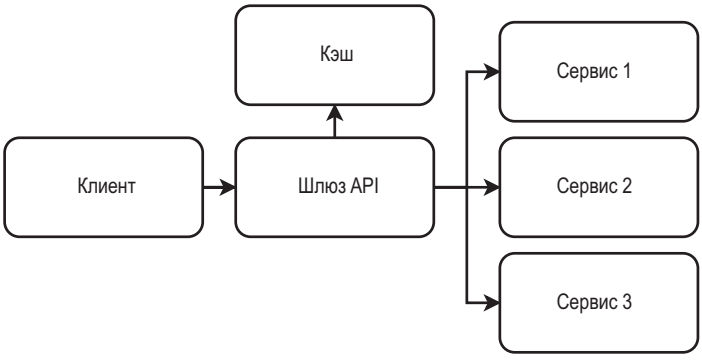

Рис. 4.9. Кэширование на уровне шлюза API. Диаграмма взята из Rob Vettor, David Coulter, Genevieve Warren. 17 мая 2020 г. Caching in a cloud-native application, Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/azure-caching

4.8.1. Стратегии чтения

Стратегии чтения оптимизируются для быстрого чтения.

Параллельный кэш (отложенная загрузка)

Паттерн Cache-aside («Параллельный кэш») обозначает размещение кэша «рядом» с базой данных. На рис. 4.10 изображена схема этого паттерна. Совершая запрос на чтение, приложение сначала обращается к кэшу, который возвращает данные при попадании. В случае промаха приложение направляет запрос на чтение к базе данных, а затем записывает данные в кэш, чтобы последующие запросы этих данных попадали в кэш. Таким образом, данные загружаются только при первом чтении — это называется *отложенной*, или *ленивой* (lazy), *загрузкой*.

Параллельный кэш лучше всего подходит для нагрузок с интенсивным чтением. Его преимущества:

- Минимизирует количество запросов на чтение и потребление ресурсов. Чтобы дополнительно сократить количество запросов, приложение мо-

жет сохранить результаты нескольких запросов к базе данных как одно значение кэша (один ключ кэша для результатов нескольких запросов к базе данных).

- В кэш записываются только запрашиваемые данные, так что легко определить необходимую емкость кэша и регулировать ее по мере надобности для снижения затрат.
- Простота реализации.

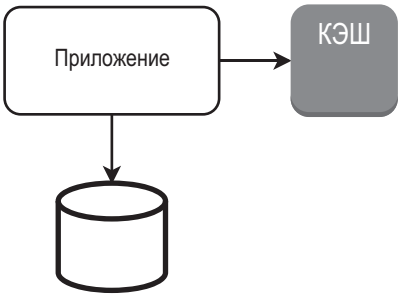

Рис. 4.10. Паттерн Cash-aside

Если кластер кэша выйдет из строя, все запросы направляются базе данных. Необходимо предусмотреть, чтобы база данных справлялась с этой нагрузкой.

Недостатки параллельного кэша:

- Кэшированные данные могут быть устаревшими/несогласованными, особенно если запись выполняется непосредственно в базу данных. Для снижения количества устаревших данных можно установить предельный срок их жизни (TTL, Time-To-Live) или использовать сквозную запись (см. ниже), чтобы каждая запись выполнялась через кэш.
- Запрос с промахом кэша медленнее запроса, обращенного непосредственно к базе данных, из-за дополнительного запроса на чтение и дополнительного запроса на запись в кэш.

Сквозное чтение

При *сквозном чтении* (read-through), *сквозной записи* (write-through) или кэшировании *с отложенной записью* (write-back) приложение направляет запросы к кэшу, который может совершать запросы к базе данных при необходимости.

На рис. 4.11 представлена архитектура сквозного чтения, сквозной записи или кэширования с отложенной записью. При промахе кэша для сквозного чтения кэш выдает запрос к базе данных и сохраняет данные в кэше (отложенная загрузка, как при параллельном кэшировании), после чего возвращает данные приложению.

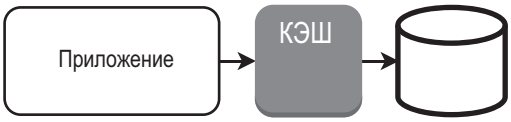

Рис. 4.11. При сквозном чтении, сквозной записи или кэшировании с отложенной записью приложение направляет запросы к кэшу, который может совершать запросы к базе данных при необходимости. Таким образом, эта простая архитектурная диаграмма может представлять все три стратегии кэширования

Сквозное чтение лучше всего подходит для нагрузок с интенсивным чтением. Так как приложение не связывается с базой данных, бремя реализации запросов к базе данных смещается с приложения на кэш. Компромисс заключается в том, что, в отличие от параллельного кэша, сквозной кэш не может группировать несколько запросов к базе данных в одно значение кэша.

4.8.2. Стратегии записи

Стратегии записи оптимизируются для минимизации устаревания кэша за счет повышения задержки или сложности.

Сквозная запись (write-through)

Каждая операция записи проходит через кэш, а затем выполняется в базе данных. Преимущества сквозной записи:

- Согласованность. Кэш никогда не устаревает, потому что данные кэша обновляются с каждой записью в базу данных.

Недостатки сквозной записи:

- Снижение скорости, так как каждая запись выполняется как в кэш, так и в базу данных.
- Проблема «холодного старта», поскольку новый узел кэша содержит отсутствующие данные, что приводит к промахам кэша. Для решения этой проблемы можно использовать параллельный кэш.
- Большинство данных никогда не читается, что создает необязательные затраты. Для сокращения неэффективного использования памяти можно настроить предельный срок жизни данных.
- Если размер кэша меньше размера базы данных, необходимо определить наиболее подходящую политику вытеснения из кэша.

Отложенная запись (write-back/write-behind)

Приложение записывает данные в кэш, но кэш записывает их в базу данных не сразу, а периодически. Преимущества отложенной записи:

- Запись в среднем выполняется быстрее, чем при сквозной записи. Запись в базу данных выполняется без блокирования.

Недостатки отложенной записи:

- Такие же, как у сквозной записи, кроме снижения скорости.
- Повышение сложности из-за того, что кэш должен обладать высокой доступностью, а следовательно, ее не получится снизить ради повышения производительности/снижения задержки. Дизайн усложняется, так как необходимы и высокая доступность, и высокая производительность.

Обходная запись (write-around)

При *обходной записи* приложение ведет запись только в базу данных. На рис. 4.12 обходная запись обычно объединяется с параллельным кэшированием или сквозным чтением. Приложение обновляет кэш при промахе.

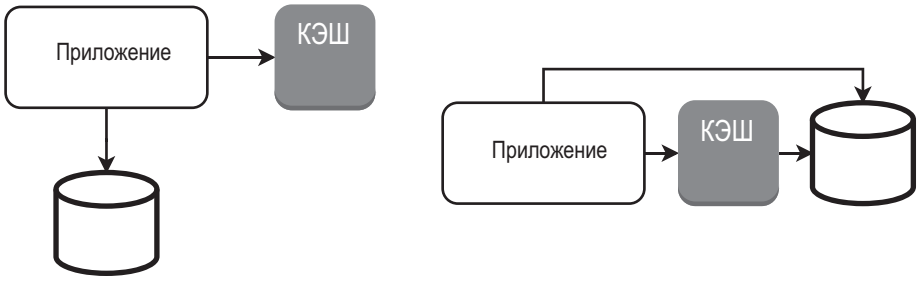

Рис. 4.12. Две возможные архитектуры обходной записи. Слева — обходная запись с параллельным кэшированием. Справа — обходная запись со сквозным кэшированием

4.9. КЭШИРОВАНИЕ КАК ОТДЕЛЬНЫЙ СЕРВИС

Почему кэширование существует как отдельный сервис? Почему бы просто не кэшировать в память хостов сервиса?

- Сервисы проектируются без сохранения состояния, так что каждый запрос назначается хосту рандомно. Так как на разных хостах могут кэшироваться разные данные, вероятность, что каждый полученный запрос будет кэширован, снижается. В этом отличие от баз данных, которые обладают состоянием и могут секционироваться, так что каждый узел базы данных с большей вероятностью будет обслуживать запросы к одним и тем же данным.
- В дополнение к сказанному выше кэширование особенно полезно при неравномерном распределении запросов, ведущих к появлению «горячих» шардов. Кэширование бесполезно, если запросы или ответы уникальны.

- Если организовать кэширование на уровне хостов, то кэш будет очищаться при каждом развертывании сервиса, что может происходить несколько раз за день.
- Кэш может масштабироваться независимо от сервисов, которые он обслуживает (хотя при этом возникают риски, рассмотренные в начале этой главы). Сервис кэширования может использовать специализированное оборудование или виртуальные машины, оптимизированные для нефункциональных требований сервиса кэширования, которые могут отличаться от требований обслуживаемых им сервисов.
- Если много клиентов одновременно отправляют один запрос, который отсутствует в кэше, сервис базы данных многократно выполнит один запрос. Кэш может устранить дубликаты и отправить один запрос сервису. Этот прием, называемый *объединением запросов*, снижает уровень трафика сервиса.

Кроме кэширования бэкенд-сервиса, стоит организовать кэширование на стороне клиентов (браузеров или мобильных приложений), чтобы по возможности избежать накладных расходов на сетевые запросы. Также следует рассмотреть возможность использования CDN.

4.10. ВИДЫ КЭШИРУЕМЫХ ДАННЫХ И СПОСОБЫ ИХ КЭШИРОВАНИЯ

Кэшироваться могут как ответы HTTP, так и запросы к базам данных. Можно кэшировать тело ответа HTTP и прочитать его по ключу кэша, который формируется из метода HTTP и URI запроса. Внутри приложения можно воспользоваться паттерном Cashe-aside для кэширования реляционных баз данных.

Кэши могут быть приватными или открытыми/общедоступными. Приватный кэш размещается на стороне клиента и используется для персонализированного контента. Общедоступный кэш находится на стороне посредника (например, CDN) или сервисов.

Какую информацию не стоит кэшировать:

- конфиденциальная информация (например, детали банковского счета) *никогда* не должна храниться в кэше;
- открытая информация в реальном времени: биржевые котировки, время прибытия рейсов, доступность номеров в отелях на ближайшее будущее и т. д.;
- платный или защищенный авторским правом контент (например, книги или платные видеоматериалы);

Общедоступную информацию, которая может изменяться, кэшировать можно, но ее необходимо сверять с сервером-источником (пример — наличие авиабилетов

или номеров в отелях на следующий месяц). Сервер отвечает кодом 304, указывающим на то, что кэшированный ответ содержит актуальную информацию, так что ответ будет намного компактнее, чем при отсутствии кэширования. При этом снижается сетевая задержка и повышается пропускная способность. Мы задаем значение `max-age` — время, в течение которого кэшированный ответ останется актуальным. Тем не менее у вас могут быть причины полагать, что в будущем возникнут некие условия, из-за которых значение `max-age` может стать слишком большим; тогда на бэкенде реализуется логика быстрой проверки актуальности кэшированного ответа. Если вы выберете это решение, в ответе необходимо возвращать `must-revalidate`, чтобы клиенты заново проверяли кэшированные ответы на бэкенде перед их использованием.

Открытую информацию, которая не будет меняться в течение длительного времени, можно кэшировать с долгим сроком жизни. Примеры — расписание автобусов или поездов.

В общем случае компания может сэкономить на оборудовании, перемещая как можно больше операций обработки и хранения данных на клиентские устройства. В этом случае датацентры используются только для хранения резервных копий критически важных данных и для взаимодействия между пользователями. Например, WhatsApp хранит подробности аутентификации пользователей и их подключений, но не хранит их сообщения (которые занимают большую часть памяти, потребляемой пользователем). Он предоставляет возможность резервного копирования Google Drive, так что затраты на резервное копирование сообщений переносятся на другую компанию. Без этих расходов WhatsApp может оставаться бесплатным для пользователей, которые платят Google за хранение данных в случае превышения порога бесплатного хранения.

Однако не следует полагаться на то, что кэширование `localStorage` будет всегда работать как задумано. Нужно учитывать вероятность кэш-промахов и подготовить сервис к получению таких запросов. Кэширование происходит на всех уровнях (клиент/браузер, балансировщик нагрузки, фронтенд/шлюз API/расширение, а также бэкенд), так что запросы проходят минимально возможное количество сервисов, что способствует снижению задержки и затрат.

Браузер начинает рендерить веб-страницу только после загрузки и обработки всех CSS-файлов последних, так что кэширование CSS браузером может значительно улучшить производительность браузерного приложения.

> **ПРИМЕЧАНИЕ** Оптимизация производительности веб-страницы за счет того, что браузер максимально быстро загружает и обрабатывает код CSS, рассматривается в статье https://csswizardry.com/2018/11/css-and-network-performance/.

Недостаток кэширования на стороне клиента заключается в том, что оно усложняет аналитику использования, так как бэкенд не получает указания на то, что клиент обратился к этим данным. Если необходимо или полезно знать,

что клиент обратился к кэшированным данным, потребуется дополнительная сложность ведения журналов использования на стороне клиента и отправки этих журналов на бэкенд.

4.11. ИНВАЛИДАЦИЯ КЭША

Содержимое кэша становится недействительным при замене или удалении элементов. Отключение (busting) кэширования — метод инвалидации кэша специально для файлов.

4.11.1. Инвалидация кэша браузера

В кэше браузера обычно задается значение `max-age` для каждого файла. Но что, если файл будет заменен новой версией до истечения его срока действия в кэше? В таких случаях используется метод «цифрового отпечатка» — присваивание этим файлам новых идентификаторов (номеров версий, имен файлов или хешей строк запросов).

Например, файл с именем style.css может быть переименован в style.b3d716.css, а хеш в имени файла может быть заменен при новом развертывании. В другом примере тег HTML `<img src=/assets/image.png />`, содержащий имя файла изображения, заменяется на `<img src=/assets/image.png?hash=a3db6e />`; для обозначения версии файла используется параметр запроса `hash`. При использовании цифрового отпечатка также можно использовать параметр управления хешем `immutable` для предотвращения необязательных запросов к серверу-источнику.

Цифровые отпечатки важны для кэширования нескольких запросов GET или файлов, зависящих друг от друга. Заголовки кэширования запросов GET не отражают взаимозависимость определенных файлов или ответов, что может привести к развертыванию старых версий файлов.

Например, обычно кэшируется код CSS и JavaScript, но не HTML, если только веб-страница не является статической; многие браузерные приложения выводят разный контент при каждом посещении. Тем не менее все они могут измениться при новом развертывании браузерного приложения. Если новая разметка HTML будет добавлена к старому CSS или JavaScript, работоспособность страницы может быть нарушена. Пользователь может инстинктивно нажать клавишу перезагрузки веб-страницы, и это решит проблему — браузер повторно проверяет сервер-источник при перезагрузке страницы. Но это плохая организация взаимодействия с пользователем. Такие проблемы трудно обнаружить в ходе тестирования. Цифровые отпечатки гарантируют, что HTML содержит верные имена файлов CSS и JavaScript.

Возможно, вы решите обойти эту проблему без цифровых отпечатков, кэшируя HTML вместе с CSS и JavaScript и установив одинаковое значение `max-age`

для всех файлов, чтобы срок их действия истек одновременно. Но браузер может выдавать запросы к этим разным файлам в разное время, с интервалами в несколько секунд. Если новое развертывание будет происходить во время выполнения этих запросов, браузер все равно может получить комбинацию старых и новых файлов.

Кроме зависимых файлов, приложение может содержать зависимые запросы GET. Например, пользователь может выдать запрос GET для списка элементов (товаров, участвующих в распродаже, номеров в отеле, рейсов в Сан-Франциско, миниатюр фотографий и т. д.), за которым следует запрос GET для подробной информации об элементе. Кэширование первого запроса может привести к запросу подробной информации о несуществующем товаре. Лучшие практики архитектуры REST рекомендуют кэшировать запросы по умолчанию, но исходя из указанных соображений можно либо отказаться от кэширования, либо установить короткий срок его действия.

4.11.2. Инвалидация кэша в сервисах кэширования

У вас нет прямого доступа к кэшам клиентов, и это ограничивает стратегии инвалидации кэша такими методами, как назначение `max-age` или цифровые отпечатки. Тем не менее можно напрямую создавать, заменять или удалять элементы в сервисе кэширования. В интернете много ресурсов, посвященных политикам замены кэшей, а их реализации выходят за рамки этой книги, поэтому мы лишь кратко перечислим наиболее распространенные способы.

- Случайная замена: при заполнении кэша заменяется случайный элемент. Это самая простая стратегия.
- Алгоритм LRU (Least Recently Used, «наиболее давно использованные»): первым заменяется элемент, с момента использования которого прошло больше всего времени.
- FIFO (First In First Out, «первым пришел, первым вышел»): элементы заменяются в порядке их добавления, независимо от частоты их использования/обращений к ним.
- LIFO (Last In First Out, «последним пришел, первым вышел»): элементы заменяются в порядке, обратном порядку их добавления, независимо от частоты их использования/обращений к ним.

4.12. РАЗОГРЕВ КЭША

Под *разогревом кэша* (cashe warming) понимается заполнение кэша данными до первого запроса к этим данным, чтобы первый запрос элемента мог быть обслужен из кэша (вместо промаха). Разогрев кэша применяется для таких сервисов, как CDN, фронтенд- или бэкенд-сервисы, но не для кэша браузера.

Преимущество разогрева кэша в том, что первый запрос предварительно кэшированных данных имеет такую же низкую задержку, как и последующие. Однако у разогрева кэша есть много недостатков, включая следующие:

- Дополнительная сложность и затраты на реализацию разогрева кэша. Сервис кэширования может содержать тысячи хостов, и их разогрев может быть сложным и дорогостоящим делом. Затраты можно уменьшить, заполняя кэш только частично — записями, которые будут запрашиваться чаще всего. Дизайн системы разогрева кэша Netflix рассматривается в статье https://netflixtechblog.com/cache-warming-agility-for-a-stateful-service-2d3b1da82642.
- Дополнительный трафик из-за направления запросов к сервису для заполнения кэша, включая фронтенд- и бэкенд-сервисы, а также сервисы баз данных. Может оказаться, что сервис не справляется с нагрузкой разогрева кэша.
- Если пользователей в базе миллионы, задержка будет наблюдаться только для первого пользователя, обратившегося к этим данным. Ускорение может не оправдать сложность и затраты на разогрев кэша. Данные, к которым часто обращаются, будут кэшированы при первом запросе, тогда как данные с редкими обращениями не оправдывают затрат ресурсов на кэширование или разогрев кэша.
- Срок действия данных в кэше не может быть коротким, в противном случае элементы кэша могут устареть до их использования, и разогрев кэша станет пустой тратой времени. Таким образом, приходится либо задать долгий срок действия, но тогда сервис кэша увеличится и станет более затратным, чем допустимо, или же назначить разный срок действия для разных элементов, вводя дополнительную сложность и риск ошибок.

Задержка 99-го процентиля для запросов без кэширования обычно должна быть меньше 1 секунды. Даже если ослабить это требование, время не должно превышать 10 секунд. Вместо разогрева кэша можно подерживать допустимое значение 99-го процентиля для запросов, обслуживаемых без кэширования.

4.13. ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

В этой главе используется материал из книги Артура Эйсмонта «Web Scalability for Startup Engineers» (McGraw Hill, 2015).

4.13.1. Кэширование

- Кевин Кроули (Kevin Crawley). Scaling Microservices — Understanding and Implementing Cache, 22 августа 2019 г. (https://dzone.com/articles/scaling-microservices-understanding-and-implementi).

- Роб Веттор (Rob Vettor), Дэвид Колтер (David Coulter), Женевьев Уоррен (Genevieve Warren). Caching in a cloud-native application, Microsoft Docs, 17 мая 2020 г., Microsoft Docs (https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/azure-caching).
- Корнелия Дэвис (Cornelia Davis). «Cloud Native Patterns»[1] (Manning Publications, 2019) (https://jakearchibald.com/2016/caching-best-practices/).
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control.
- Том Баркер (Tom Barker). «Intelligent Caching» (O'Reilly Media, 2017).

ИТОГИ

- Проектирование сервисов с сохранением состояния сложнее и чревато большим количеством ошибок по сравнению с проектированием сервиса без сохранения состояния, поэтому дизайнеры систем стараются ограничиваться вторыми и использовать общие сервисы с сохранением состояния.
- Каждая технология хранения данных относится к определенной категории из перечисленных ниже. Вы должны их различать.
  - Базы данных SQL или NoSQL. Базы данных NoSQL можно разделить на столбцово-ориентированные и базы данных «ключ — значение».
  - Хранилища документов.
  - Графовые хранилища.
  - Файловые хранилища.
  - Блоковые хранилища.
  - Объектные хранилища.
- Выбор способа хранения данных сервиса подразумевает выбор между базой данных и другой категорией хранилища.
- Существуют разные технологии репликации для масштабирования базы данных, включая репликацию с одним лидером, репликацию с несколькими лидерами, репликацию без лидера и другие методы, — например репликацию HDFS, которую нельзя однозначно отнести ни к одной из этих категорий.
- Шардирование необходимо, если база данных превышает емкость хранилища данных одного хоста.
- Запись в базу данных обходится дорого и плохо масштабируется, поэтому операции записи следует по возможности минимизировать. Агрегирование событий помогает сократить частоту записи в базу данных.

[1] Дэвис К. «Шаблоны проектирования для облачной среды».

- Лямбда-архитектура основана на использовании параллельных пакетных и потоковых пайплайнов для обработки одних и тех же данных. Она реализует преимущества обоих подходов, одновременно позволяя им компенсировать взаимные недостатки.
- Денормализация часто используется для оптимизации задержки чтения и упрощения запросов `SELECT`. Ее оборотные стороны — ухудшение согласованности, замедление записи, необходимость большего объема памяти и замедление перестроения индексов.
- Кэширование частых запросов в памяти сокращает среднюю задержку запроса.
- Стратегии чтения ускоряют операции чтения за счет возможного устаревания содержимого кэша.
- Параллельное кэширование лучше подходит для нагрузок с интенсивным чтением, но кэшированные данные могут устареть, а промахи кэша обрабатываются медленнее, чем если бы кэш отсутствовал.
- Кэш со сквозным чтением выдает запросы к базе данных, избавляя приложение от этой необходимости.
- Кэш со сквозной записью никогда не устаревает, но работает медленнее.
- Кэш с отложенной записью периодически записывает обновленную информацию в базу данных. В отличие от других структур кэша, он должен иметь высокую доступность для предотвращения возможной потери данных из-за сбоев.
- Для кэша с обходной записью характерна медленная запись и более высокий риск устаревания кэша. Он подходит для сценариев, в которых вероятность изменения кэшированных данных невелика.
- Специализированный сервис кэширования может обслуживать пользователей намного быстрее, чем кэширование в памяти хостов сервисов.
- Не кэшируйте конфиденциальные данные. Кэшируйте открытые данные; повторная проверка и время действия данных в кэше зависят от того, как часто и с какой вероятностью изменяются данные.
- Стратегии инвалидации кэша различаются для сервисов и клиентов, потому что в первом случае вы имеете доступ к хостам, а во втором нет.
- Разогрев кэша позволяет первому пользователю кэшированных данных получить ответ так же быстро, как и последующим, но у разогрева кэша много недостатков.

5 Распределенные транзакции

В ЭТОЙ ГЛАВЕ

- ✓ Обеспечение согласованности данных между несколькими сервисами
- ✓ Использование порождения событий для обеспечения масштабируемости, доступности, согласованности и снижения затрат
- ✓ Запись изменений в разные сервисы с использованием CDC (Change Data Capture)
- ✓ Реализация транзакции: хореография и оркестрация

Единица работы в системе может включать в себя запись данных в несколько сервисов. Каждая запись в каждый сервис является отдельным запросом/событием. Любая операция записи может завершиться неудачей; причины могут быть разными, от ошибок в программах до сбоев хоста или сети. Это может привести к рассогласованию данных между сервисами. Например, если клиент купил путевку, которая включает билет на самолет и бронь номера в отеле, системе может понадобиться выполнить запись в сервис бронирования билетов, сервис бронирования номеров и сервис платежей. Если хотя бы одна операция записи завершится неудачей, система окажется в несогласованном состоянии.

Другой пример — система передачи сообщений, которая отправляет сообщения получателям и регистрирует в базе данных информацию об отправленных сообщениях. Если сообщение было успешно отправлено на устройство получателя, но запись в базу данных завершилась неудачей, для системы все выглядит так, словно сообщение не было доставлено.

Транзакция представляет собой механизм группировки нескольких операций чтения и записи в логическую единицу для сохранения согласованности данных между сервисами. Эти операции выполняются атомарно как единая операция, и вся транзакция завершается либо полным успехом (закрепление), либо неудачей (отмена, откат). Транзакция обладает набором свойств ACID, хотя разные базы данных по-разному трактуют концепции ACID, так что реализации тоже различаются.

Если для распределения операций записи вы можете воспользоваться платформой потоковой передачи событий (такой, как Kafka), чтобы нижележащие сервисы получали записи по модели pull, а не по модели push, рекомендуется поступить именно так (сравнение моделей pull и push представлено в разделе 4.6.2). Для других сценариев рассмотрим концепцию *распределенных транзакций*, объединяющую отдельные запросы записи в одну распределенную (атомарную) транзакцию. Также рассмотрим концепцию *консенсуса* — когда все сервисы соглашаются с тем, что событие записи произошло (или не произошло). Для обеспечения согласованности между сервисами необходим консенсус, несмотря на возможные отказы при событиях записи. В этом разделе описаны алгоритмы поддержания согласованности в распределенных транзакциях:

- взаимосвязанные концепции порождения событий, CDC (Change Data Capture) и EDA (Event Driven Architecture);
- контрольные точки и очереди недоставленных сообщений рассматривались в разделах 3.3.6 и 3.3.7;
- сага;
- двухфазный коммит. (Эта тема выходит за рамки книги. Кратко она рассмотрена в приложении Г.)

Двухфазный коммит и сага достигают консенсуса (все операции или фиксируются, или принудительно прерываются), тогда как другие методы назначают выбранную базу данных источником достоверной информации на случай, если сбои при записи приведут к рассогласованию данных.

5.1. СОБЫТИЙНАЯ АРХИТЕКТУРА (EDA)

В своей книге «Scalability for Startup Engineers» Артур Эйсмонт пишет: «Событийная архитектура (EDA, Event-Driven Architecture) — архитектурный стиль, в котором большинство взаимодействий между компонентами реализуется

путем публикации событий, которые уже произошли, вместо запросов на выполнение работы».

Стиль EDA является асинхронным и неблокирующим. Запрос не нуждается в обработке, которая может занять значительное время и создать высокую задержку. Вместо этого достаточно опубликовать событие. Если событие успешно опубликовано, сервис возвращает ответ с признаком успеха. Событие может быть обработано в будущем. При необходимости сервер может отправить ответ источнику запроса. EDA способствует слабой связанности (loose coupling), масштабируемости и высокой скорости получения ответов (низкая задержка).

Альтернатива EDA — выдача сервисом запросов, обращенных напрямую к другому сервису. Независимо от того, был ли такой запрос блокирующим или неблокирующим, недоступность или низкая производительность любого из сервисов будет означать, что система в целом недоступна. Запросу также выделяется программный поток в каждом сервисе, так что на время обработки запроса количество доступных потоков уменьшается на единицу. Это особенно заметно, если обработка запроса занимает много времени или если она выполняется при выбросах трафика. Выброс трафика может перегрузить сервис и привести к тайм-аутам 504. Такое решение повлияет на источники запросов, поскольку каждый из них должен продолжать поддерживать программный поток, пока запрос не завершен, так что у устройства источника запроса останется меньше ресурсов для другой работы.

Чтобы выбросы трафика не приводили к сбоям, придется применять сложные решения с автомасштабированием или поддерживать большие кластеры хостов, что создает дополнительные затраты. (Другое возможное решение — ограничение частоты — рассматривается в главе 8.)

Все эти альтернативы затратны, сложны и ненадежны и хуже масштабируются. Сильная согласованность и низкая задержка, которые ими обеспечиваются, могут оказаться ненужными пользователям.

Менее ресурсоемкий подход основан на публикации событий в журнале событий. Сервису-публикатору не обязательно постоянно потреблять поток и ожидать сервис-подписчик для завершения обработки события.

На практике можно полностью отказаться от неблокирующей философии EDA, например, выполнением проверки запроса при его выдаче. Допустим, сервис может проверить, что в запросе заполнены все обязательные поля, а значения действительны; строковое поле должно быть непустым и отличным от `null`; оно также может иметь минимальную и максимальную длину. Вы можете выбрать этот вариант, чтобы недействительный запрос был обнаружен быстро и ресурсы и время не тратились на сохранение недействительных данных только для того, чтобы обнаружить ошибку позже. Порождение событий и CDC (Change Data Capture, «отслеживание измененных данных») являются примерами EDA.

5.2. ПОРОЖДЕНИЕ СОБЫТИЙ

Порождение событий — паттерн хранения данных или изменений в данных как событий в журнале, поддерживающем только добавление данных. Согласно Корнелии Дэвис («Cloud Native Patterns»), идея порождения событий заключается в том, что журнал событий является источником достоверных данных, а все остальные базы данных — проекциями журнала событий. Любая запись должна сначала выполняться в журнале событий. После того как эта запись завершается успехом, один или несколько обработчиков событий потребляют новое событие и записывают его в другие базы данных.

Порождение событий не привязано к конкретному источнику данных. Оно может сохранять события из разных источников, например взаимодействий с пользователем или внешних и внутренних систем. На рис. 5.1 порождение событий состоит из публикации и сохранения детализированных, изменяющих состояние событий некоторой сущности в виде последовательности событий. Эти события хранятся в журнале, а подписчики обрабатывают события из журнала для определения текущего состояния сущности. Следовательно, сервис издателя асинхронно взаимодействует с сервисом подписчика через журнал событий.

Рис. 5.1. В схеме порождения событий издатель публикует последовательность событий в журнале, отмечающем изменения в состоянии сущности. Подписчик последовательно обрабатывает события в журнале, чтобы определить текущее состояние сущности

Эта схему можно реализовать разными способами. Издатель может опубликовать событие в хранилище событий или в журнале, поддерживающем только добавление данных (например, в топике Kafka), записать строку в реляционную БД (SQL), записать документ в документную БД (такую, как MongoDB или Couchbase) или даже записать данные в БД в памяти (например, Redis или Apache Ignite) для обеспечения низкой задержки.

> **ВОПРОС** Что, если на хосте подписчика произойдет сбой в ходе обработки события? Как сервис-подписчик узнает, что событие нужно обработать снова?

Порождение событий предоставляет полный след аудита для всех событий в системе, а также возможность получать информацию о прошлых состояниях системы посредством воспроизведения событий для отладки или аналитики. Порождение событий также позволяет изменять бизнес-логику посредством

введения новых типов событий и обработчиков, не влияя на существующие данные.

Порождение событий добавляет сложность в проектирование и разработку системы, поскольку приходится управлять хранилищами событий, воспроизведением, версионированием и эволюцией схем. Оно повышает требования к объему хранилища данных. Воспроизведение событий становится более затратным и длительным с ростом журналов.

5.3. CDC

Суть CDC (Change Data Capture, «отслеживание измененных данных») — это регистрация событий изменения данных в потоке событий журнала изменений и предоставление потока событий через API.

Схема работы CDC представлена на рис. 5.2. Одно изменение или группа изменений могут публиковаться в потоке событий журнала изменений как одно событие. Этот поток событий имеет несколько потребителей, каждый из которых соответствует сервису/приложению/базе данных. Каждый потребитель потребляет событие и передает его нижележащему сервису для обработки.

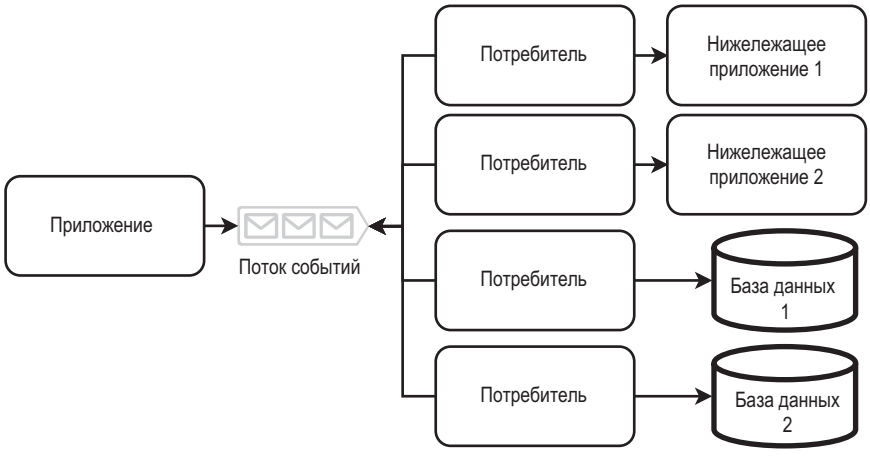

Рис. 5.2. Использование потока событий журнала изменений для синхронизации изменений данных. Кроме потребителей, бессерверные функции могут использоваться для передачи изменений к нижележащим сервисам или базам данных

CDC обеспечивает согласованность и более низкую задержку по сравнению с порождением событий. Каждый запрос обрабатывается почти в реальном времени — в отличие от порождения событий, при котором запрос может оставаться в журнале какое-то время, прежде чем будет обработан подписчиком.

Паттерн Transaction log tailing («Отслеживание журнала транзакций») (Крис Ричардсон (Chris Richardson), «Microservices Patterns: With Examples in Java»[1], Manning Publications, 2019) — еще один паттерн проектирования систем для предотвращения возможного рассогласования, когда процесс должен произвести запись в базу данных и передать событие в Kafka. Одна из двух операций записи может завершиться неудачей, что приведет к рассогласованию.

На рис. 5.3 изображена схема паттерна Transaction log tailing. В этом паттерне процесс, называемый *майнером*, читает журнал транзакций базы данных и передает каждое обновление в виде события.

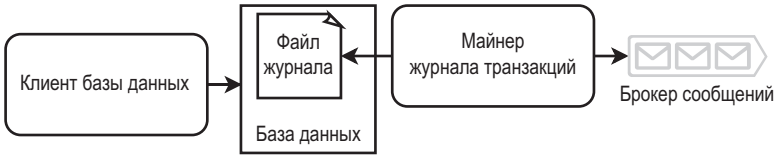

Рис. 5.3. Паттерн Transaction log tailing. Сервис отправляет запрос на запись в базу данных, которая записывает этот запрос в свой файл журнала. Майнер журнала транзакций читает файл журнала и принимает этот запрос, а затем отправляет событие брокеру сообщений

К числу платформ CDC относятся Debezium (https://debezium.io/), Databus (https://github.com/linkedin/databus), DynamoDB Streams (https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html) и Eventuate CDC Service (https://github.com/eventuate-foundation/eventuate-cdc). Они могут использоваться в качестве майнеров журнала транзакций.

Майнеры журналов транзакций могут генерировать дубликаты событий. Один из способов обработки дубликатов событий основан на использовании механизмов брокера сообщений, которые гарантируют, что событие будет доставлено ровно один раз. Другое возможное решение — идемпотентное определение и обработка событий.

5.4. СРАВНЕНИЕ ПОРОЖДЕНИЯ СОБЫТИЙ И CDC

Событийная архитектура (EDA), порождение событий и CDC — взаимосвязанные концепции, используемые в распределенных системах для распространения изменений данных среди заинтересованных потребителей и нижележащих сервисов. Они ослабляют связи между сервисами, используя паттерны асинхронных коммуникаций для распространения информации об этих изменениях. В дизайне некоторых систем порождение событий может использоваться вместе с CDC. Например, вы можете использовать порождение событий внутри сервиса для

[1] Ричардсон К. «Микросервисы. Паттерны разработки и рефакторинга». СПб., издательство «Питер».

записи изменений данных в виде событий, одновременно используя CDC для направления этих событий к другим сервисам. Они различаются по целям, по степени детализации и по источникам достоверной информации. Краткая сводка этих различий приведена в табл. 5.1.

Таблица 5.1. Различия между порождением событий и CDC

	Порождение событий	**CDC (отслеживание измененных данных)**
Назначение	Запись событий как источника достоверной информации	Синхронизация изменений данных посредством направления событий от сервиса-источника к нижележащим сервисам
Источник достоверной информации	Журнал (или события, опубликованные в журнале) является источником достоверной информации	База данных в сервисе-издателе. Опубликованные события источником достоверной информации не являются
Детализация	Детализированные события, представляющие конкретные действия или изменения в состоянии	Отдельные изменения на уровне базы данных, например создание, обновление или удаление строк или документов

5.5. СУПЕРВИЗОР ТРАНЗАКЦИЙ

Супервизор транзакций представляет собой процесс, который гарантирует, что транзакция будет либо успешно завершена, либо компенсирована. Он может быть реализован как периодическое пакетное задание либо как бессерверная функция. На рис. 5.4 показан пример супервизора транзакций.

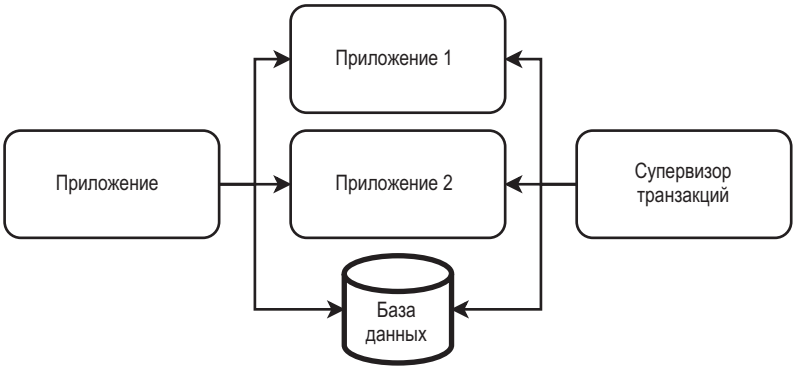

Рис. 5.4. Супервизор транзакций. Приложение может выполнять запись в множественные нижележащие приложения и базы данных. Супервизор транзакций периодически синхронизирует различные приемники на случай неудачных операций записи

Как правило, супервизор транзакций сначала реализуется как интерфейс для ручного анализа несогласованностей и ручного выполнения компенсирующих транзакций. Автоматический откат транзакций в общем случае рискован, и к нему следует подходить с осторожностью. Прежде чем автоматизировать компенсирующую транзакцию, ее необходимо тщательно протестировать. Также убедитесь в отсутствии любых других механизмов распределенных транзакций; в противном случае они будут мешать друг другу, что приведет к потере данных или затруднению отладки.

Компенсирующая транзакция всегда должна регистрироваться независимо от того, выполнялась она вручную или автоматически.

5.6. САГА

Сага (saga) представляет собой долгосрочную транзакцию, которая записывается в виде последовательности транзакций. Все транзакции должны завершиться успешно; в противном случае будет выполнена компенсирующая транзакция для отката выполняемых транзакций. Сага является паттерном, упрощающим управление сбоями. Сама по себе сага не имеет состояния.

Типичная реализация саги подразумевает взаимодействие сервисов через брокер сообщений, такой как Kafka или RabbitMQ. В этой книге для работы с сагой используется Kafka.

Важный пример использования саги — выполнение распределенной транзакции только в том случае, если определенные сервисы удовлетворяют определенным запросам. Например, при бронировании туристической путевки сервис может отправить один запрос на запись в сервис бронирования авиабилетов, а другой — в сервис бронирования номера в отеле. Если доступных билетов или номеров нет, то необходимо откатить всю сагу.

Возможно, сервисам бронирования билетов и отелей также понадобится выполнить запись в сервис платежей, который существует отдельно от сервисов билетов и отелей; это объясняется следующими причинами:

- Сервис платежей не должен обрабатывать платежи до тех пор, пока сервис авиабилетов не подтвердит доступность билета, а сервис отелей — доступность номера. В противном случае может оказаться, что деньги будут списаны еще до подтверждения всего тура.
- Сервисы авиабилетов и отелей могут принадлежать другим компаниям, и мы не можем передавать им конфиденциальную платежную информацию пользователей. В этом случае наша компания должна обработать платеж пользователя и выполнить платежи, предназначенные для других компаний.

Если транзакция к сервису платежей завершится неудачей, всю сагу необходимо откатить в обратном порядке, используя компенсирующие транзакции с двумя другими сервисами.

Существуют два способа структурирования координации: хореография (параллельный) и оркестрация (линейный). В оставшейся части этого раздела обсуждается один пример хореографии и один пример оркестрации, после чего хореография сравнивается с оркестрацией. Другой пример рассматривается здесь: https://microservices.io/patterns/data/saga.html.

5.6.1. Хореография

В случае хореографии сервис, начинающий сагу, взаимодействует с двумя топиками Kafka. Он направляет событие в один топик Kafka, чтобы начать распределенную транзакцию, и потребляет событие из другого топика Kafka для выполнения завершающей логики. Другие сервисы в саге взаимодействуют непосредственно друг с другом через топики Kafka.

На рис. 5.5 представлена хореографическая сага для бронирования тура. В этой главе на диаграммах, где присутствуют топики Kafka, потребление событий обозначено стрелками, ведущими от топика. В остальных главах книги потребление событий обозначается стрелками, указывающими на топик. Дело в том, что если придерживаться обозначений, которые использовались в других главах, это создаст путаницу. Диаграммы из этой главы иллюстрируют множественные сервисы, потребляющие события из нескольких топиков и направляющие их в другие топики, поэтому выбрано именно такое направление стрелок.

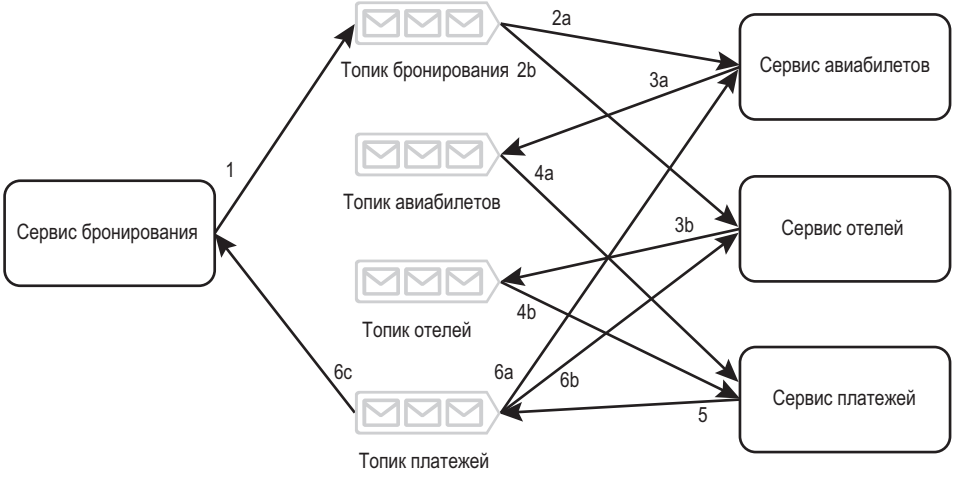

Рис. 5.5. Хореографическая сага для бронирования авиабилетов и номеров в отеле. Две метки с одинаковыми числами, но разными буквами представляют шаги, выполняемые параллельно

Последовательность действий успешного бронирования выглядит так:

1. Пользователь может отправить запрос на бронирование в сервис бронирования. Сервис направляет событие запроса к топику бронирования.
2. Сервисы билетов и отелей потребляют это событие запроса на бронирование. Они должны подтвердить, что их запросы могут быть выполнены. Оба сервиса могут записать это событие в свои базы данных, с идентификатором бронирования и состоянием `AWAITING_PAYMENT`.
3. Оба сервиса — сервис билетов и сервис отелей — направляют событие запроса платежа топику билетов и топику отелей соответственно.
4. Сервис платежей потребляет эти события запросов платежей из топика билетов и топика отелей. Так как эти два события потребляются в разное время и, скорее всего, разными хостами, сервис платежей должен сохранить подтверждение этих событий в базе данных, так что хосты сервисов будут знать о получении всех необходимых событий. Когда все необходимые события будут получены, сервис платежей обработает платеж.
5. Если платеж прошел успешно, то сервис платежей направляет событие успешного платежа топику платежа.
6. Сервис билетов, сервис отелей и сервис бронирования потребляют это событие. Сервис билетов и сервис отелей подтверждают бронирование, что может включать изменение состояния идентификатора бронирования на `CONFIRMED` или другую логику обработки и бизнес-логику по мере необходимости. Сервис бронирования информирует пользователя о том, что бронирование подтверждено.

Шаги 1–4 являются компенсируемыми транзакциями, которые могут откатываться посредством компенсации транзакций. Шаг 5 является поворотной транзакцией. Транзакции после поворотной транзакции можно повторять, пока они не завершатся успехом. Транзакции шага 6 могут повторяться; это пример CDC, который обсуждается в разделе 5.3. Сервису бронирования не нужно ждать ответов от сервиса билетов или сервиса отелей.

Можно задать вопрос: как внешняя компания подписывается на топик Kafka вашей компании? Ответ: она этого не делает. По причинам безопасности прямой внешний доступ к сервису Kafka никогда не предоставляется. Мы упростили подробности для лучшего понимания. Сервис билетов и сервис отелей в действительности принадлежат вашей компании. Они взаимодействуют напрямую с сервисом/топиком и направляют запросы к внешним сервисам. На рис. 5.5 эти подробности не показаны, чтобы не перегружать схему дизайна.

Если сервис платежа отвечает ошибкой, указывающей на то, что билет невозможно забронировать (возможно, потому, что указанный рейс полностью выкуплен или отменен), шаг 6 будет другим. Вместо того чтобы подтверждать

бронирование, сервис билетов и сервис отелей отменяют его, и сервис бронирования возвращает пользователю соответствующий ответ с ошибкой. Компенсация транзакций, выполняемых по ответу с ошибкой от сервиса отелей или платежей, выполняется аналогично, поэтому они здесь не рассматриваются.

О чем еще стоит упомянуть в отношении хореографии:

- отсутствие двусторонних линий, то есть сервис не может производить и потреблять события в одном топике;
- никакие два сервиса не производят события в одном топике;
- сервис может подписаться на несколько топиков. Если сервис должен получать несколько событий из нескольких топиков, то прежде чем он сможет выполнить действие, он должен записать в базу данных информацию о том, что он получил некоторые события; тогда он сможет прочитать базу данных и определить, получены ли все необходимые события;
- между топиками и сервисами могут существовать отношения «один ко многим» или «многие к одному», но не «многие ко многим»;
- хореография может включать циклы. Обратите внимание на циклы на рис. 5.5 (топик отелей > сервис платежей > топик платежей > сервис отелей > топик отелей).

На рис. 5.5 мы видим множество линий между разными топиками и сервисами. Хореография между большим количеством топиков и сервисов может быть сложной, ненадежной и трудной в обслуживании.

5.6.2. Оркестрация

В случае оркестрации сервис, начинающий сагу, является оркестратором. Оркестратор взаимодействует с каждым сервисом через топик Kafka. На каждом шаге саги оркестратор должен направлять события в топик, чтобы запросить начало этого шага, и потреблять из другого топика, чтобы получить результат этого шага.

Оркестратор представляет собой машину с конечным числом состояний, которая реагирует на события и выдает команды. Оркестратор должен содержать только последовательность шагов. Он не должен содержать никакую другую бизнес-логику, кроме механизма компенсации.

На рис. 5.6 изображена оркестрованная сага для бронирования тура. Шаги успешного процесса бронирования перечислены ниже.

1. Оркестратор направляет событие запроса билета топику бронирования.
2. Сервис билетов потребляет событие запроса билета и бронирует для заданного идентификатора бронирования билет на авиарейс с состоянием `AWAITING_PAYMENT`.

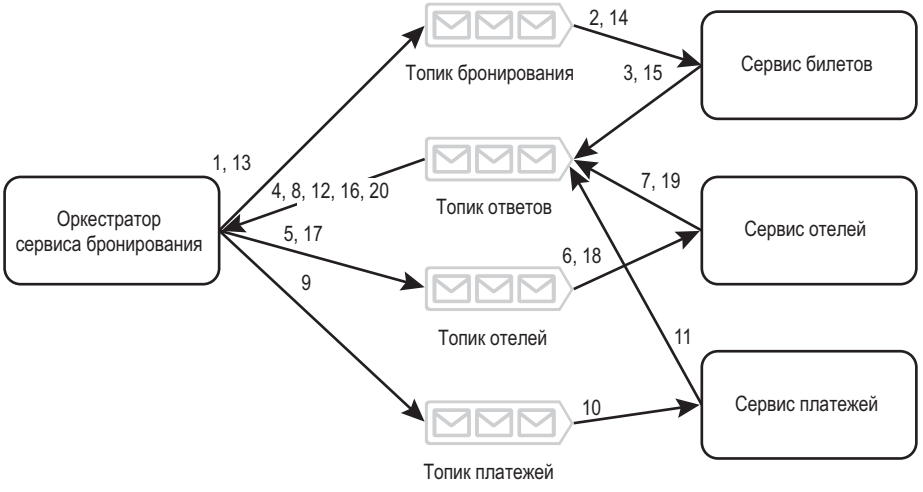

Рис. 5.6. Оркестрованная сага для бронирования билета и отеля для тура

3. Сервис билетов направляет событие «билет ожидает платежа» топику ответов.
4. Оркестратор потребляет событие «билет ожидает платежа».
5. Оркестратор направляет событие запроса на бронирование номера топику отелей.
6. Сервис отелей потребляет событие запроса на бронирование номера и резервирует номер для заданного идентификатора бронирования с состоянием `AWAITING_PAYMENT`.
7. Сервис отелей направляет событие «номер ожидает платежа» топику ответов.
8. Оркестратор потребляет событие «номер ожидает платежа».
9. Оркестратор направляет событие запроса платежа топику платежей.
10. Сервис платежей потребляет событие запроса платежа.
11. Сервис платежей обрабатывает платеж, после чего направляет событие подтверждения платежей топику ответов.
12. Оркестратор потребляет событие подтверждения платежа.
13. Оркестратор направляет событие подтверждения платежа топику бронирования.
14. Сервис билетов потребляет событие подтверждения платежа и изменяет состояние на `CONFIRMED`, подтверждая бронирование.
15. Сервис билетов направляет событие подтверждения билета топику ответов.

16. Оркестратор потребляет это событие подтверждения билета из топика ответов.
17. Оркестратор направляет событие подтверждения платежа топику отелей.
18. Сервис номеров потребляет это событие подтверждения события и изменяет состояние на `CONFIRMED`, подтверждая бронирование.
19. Сервис отелей направляет событие подтверждения номера топику ответов.
20. Оркестратор сервиса бронирования потребляет событие подтверждения номера. Затем он выполняет следующие шаги, такие как отправка успешного ответа пользователю или выполнение любой логики, внутренней для сервиса бронирования.

Шаги 18 и 19 кажутся лишними, так как на шаге 18 сбоя не будет; он может повторять попытки, пока не получит успешный результат. Шаги 18 и 20 могут выполняться параллельно. Но мы выполняем эти шаги линейно для сохранения единства подхода.

Шаги 1–13 являются компенсируемыми транзакциями. Шаг 14 является поворотной транзакцией. Шаги 15 и далее являются транзакциями с возможностью повторения.

Если хотя бы один из трех сервисов отправляет ответ с ошибкой топику бронирования, оркестратор может направлять события другим сервисам для запуска компенсирующих транзакций.

5.6.3. Сравнение

В табл. 5.2 приведено сравнение хореографии с оркестрацией. Вы должны знать различия между ними, а также их компромиссы, чтобы понять, какой вариант лучше подходит для конкретного дизайна системы. Итоговое решение может быть отчасти субъективным, но знание различий также поможет понять, что вы приобретаете или теряете, выбирая тот или иной подход.

Таблица 5.2. Хореографическая сага и оркестрованная сага

Хореография	Оркестрация
Запросы к сервисам направляются параллельно. Это объектно-ориентированный паттерн «Наблюдатель»	Запросы к сервисам направляются линейно. Это объектно-ориентированный паттерн «Контроллер»
Сервис, начинающий сагу, взаимодействует с двумя топиками Kafka. Он направляет события в один топик Kafka для запуска распределенной транзакции и потребляет из другого топика Kafka для выполнения завершающей логики	Оркестратор взаимодействует с каждым сервисом через топик Kafka. На каждом шаге саги оркестратор должен направлять события в топик, чтобы запросить запуск этого шага, и потреблять из другого топика, чтобы получить результат этого шага

Таблица 5.2 (*окончание*)

Хореография	Оркестрация
Сервис, начавший сагу, содержит только код, который направляет события в первый топик саги и потребляет из последнего топика саги. Разработчик должен прочитать код каждого сервиса, задействованного в саге, чтобы понять его действия	Оркестратор содержит код отправки и потребления, соответствующий шагам саги. Таким образом, чтение кода оркестратора позволит понять сервисы и все шаги распределенной транзакции
Сервису, например, бухгалтерскому сервису на рис. 5.5 в книге Ричардсона, может потребоваться подписаться на несколько топиков Kafka. Дело в том, что он может отправить некоторое событие только при потреблении определенных событий от нескольких сервисов. Это означает, что он должен выполнить запись в базу данных, события которой он уже потребил	Кроме оркестратора, каждый сервис подписывается только на один топик Kafka (от еще одного сервиса). Отношения между разными сервисами проще понять. В отличие от хореографии, сервису никогда не требуется потреблять несколько событий от разных сервисов, прежде чем он сможет отправить определенное событие, так что это позволит сократить количество операций записи в базу данных
Расходует меньше ресурсов, требует меньшего объема передаваемой информации и меньшего сетевого трафика; как следствие, имеет меньшую общую задержку	Так как каждый шаг должен пройти через оркестрацию, событий будет вдвое больше, чем при хореографии. Общий эффект выражается в том, что оркестрация расходует больше ресурсов, передает больше информации и генерирует больше сетевого трафика; как следствие, увеличивается общая задержка
Параллельные запросы также способствуют снижению задержки	Запросы линейны, поэтому задержка выше
Сервисы имеют менее независимый цикл разработки, потому что разработчики должны понимать все сервисы для изменения любого из них	Сервисы более независимы. Изменение в сервисе влияет только на оркестратор и не влияет на другие сервисы
Отсутствие единой точки сбоя, как при оркестрации (то есть ни один сервис не должен обладать высокой доступностью, кроме сервиса Kafka)	Если в сервисе оркестрации происходит сбой, вся сага не может быть выполнена (то есть оркестратор и сервис Kafka должны иметь высокую доступность)
Компенсирующие транзакции инициируются различными сервисами, задействованными в саге	Компенсирующие транзакции инициируются оркестратором

5.7. ДРУГИЕ ТИПЫ ТРАНЗАКЦИЙ

Следующие консенсусные алгоритмы обычно более полезны для достижения консенсуса на большом количестве узлов и, как правило, в распределенных базах

данных. В книге они обсуждаться не будут. Подробно они рассмотрены в книге Мартина Клеппмана «Designing Data-Intensive Applications»:

- запись с кворумом;
- Paxos и EPaxos;
- Raft;
- Zab (протокол атомарной рассылки ZooKeeper) — используется Apache ZooKeeper.

5.8. ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

- Мартин Клеппман. «Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems»[1] (O'Reilly Media, 2017).
- Борис Сколл (Boris Scholl), Трент Свенсон (Trent Swanson), Питер Джасовец (Peter Jausovec). «Cloud Native: Using Containers, Functions, and Data to Build Next-Generation Applications» (O'Reilly Media, 2019).
- Корнелия Дэвис. «Cloud Native Patterns» (Manning Publications, 2019).
- Крис Ричардсон. «Microservices Patterns: With Examples in Java»[2] (Manning Publications, 2019). В главе 3.3.7 обсуждается паттерн Transaction log tailing. В главе 4 рассматривается концепция саги.

ИТОГИ

- Распределенная транзакция записывает одни данные в несколько сервисов, с согласованностью в конечном счете или консенсусом.
- При порождении событий события записи сохраняются в журнале, который становится источником достоверной информации и следом аудита, позволяющим воспроизвести события для реконструкции состояния системы.
- В концепции CDC поток событий имеет несколько потребителей, каждый из которых соответствует нижележащему сервису.
- Сага представляет собой серию транзакций, которые либо целиком завершаются успешно, либо целиком откатываются.
- Хореография (параллельный) и оркестрация (линейный) — два способа координации саг.

---

[1] Клеппман М. «Высоконагруженные приложения. Программирование, масштабирование, поддержка». СПб., издательство «Питер».

[2] Ричардсон К. «Микросервисы. Паттерны разработки и рефакторинга». СПб., издательство «Питер».

6 Популярные сервисы для функционального секционирования

В ЭТОЙ ГЛАВЕ

- ✓ Централизация сквозной функциональности с использованием шлюзов API или сервисной сети / паттерна sidecar
- ✓ Минимизация сетевого трафика с помощью сервисов метаданных
- ✓ Веб- и мобильные фреймворки для выполнения требований
- ✓ Реализация функциональности в виде библиотек или сервисов
- ✓ Выбор парадигмы API между REST, RPC и GraphQL

Ранее в книге мы обсуждали функциональное секционирование как метод масштабируемости, отделяющий конкретные функции от бэкенда для выполнения на выделенных кластерах. В этой главе мы сначала обсудим шлюз API, затем паттерн sidecar (также называемый сервисной сетью), появившийся относительно недавно. Затем поговорим о централизации общих данных в сервис метаданных. У всех этих сервисов есть одно общее свойство: они содержат функциональность, общую для многих сервисов бэкенда, которую можно отделить от этих сервисов и представить в виде общих сервисов.

ПРИМЕЧАНИЕ Первая рабочая версия Istio — популярной реализации сервисной сети — была выпущена в 2018 году.

Наконец, мы обсудим фреймворки, которые могут использоваться для разработки различных компонентов при проектировании систем.

6.1. ОБЩАЯ ФУНКЦИОНАЛЬНОСТЬ РАЗНЫХ СЕРВИСОВ

Сервис может иметь много нефункциональных требований, причем сервисы с разными функциональными требованиями могут иметь общие нефункциональные. Например, сервис для вычисления налога с продаж и сервис для проверки свободных номеров в отелях могут пользоваться кэшированием для повышения производительности или принимать запросы только от зарегистрированных пользователей.

Реализация этих функций отдельно для каждого сервиса может привести к дублированию работы или кода. Повышается вероятность ошибок и снижается эффективность, так как драгоценные инженерные ресурсы будут заняты большим объемом работы.

Одно из возможных решений — поместить код в библиотеки, которые могут использоваться разными сервисами. Однако у такого решения есть недостатки, описанные в разделе 6.7. Обновлениями библиотек управляют пользователи, так что сервисы могут использовать старые версии с ошибками или дефектами безопасности, исправленными в новых версиях. На каждом хосте, на котором выполняется сервис, выполняются и библиотеки, поэтому разные функции не могут масштабироваться независимо.

Проблема решается централизацией сквозной функциональности через *шлюз API*. Шлюз API представляет собой облегченный веб-сервис, включающий машины без состояния, находящиеся в нескольких датацентрах. Он предоставляет общую функциональность многочисленных сервисов организации; таким образом реализуется централизация сквозной функциональности разных сервисов, даже написанных на разных языках программирования. Решение должно быть максимально простым, несмотря на его многочисленные обязанности. Примеры шлюзов API, предоставляемых облачными платформами, — Amazon API Gateway (https://aws.amazon.com/api-gateway/) и Kong (https://konghq.com/kong).

Ниже перечислены основные функции шлюза API, которые можно разделить на категории.

6.1.1. Безопасность

Эти функции предотвращают несанкционированный доступ к данным сервиса:

- *Аутентификация*: проверяет, что запрос поступил от авторизованного пользователя.

- *Авторизация*: проверяет, что пользователю разрешено отправить запрос.
- *Завершение SSL-запросов*: завершение обычно обеспечивается не самим шлюзом API, а отдельным прокси-сервером HTTP, который выполняется в виде процесса на том же хосте. Завершение происходит на шлюзе API, потому что на балансировщике нагрузки это обходится слишком дорого. Хотя обычно говорят «завершение SSL», на практике используется протокол TLS, пришедший на смену SSL.
- *Шифрование данных на стороне сервера*: если вам понадобится безопасно хранить данные на хостах бэкенда или в базе данных, шлюз API может шифровать данные перед сохранением и дешифровать их перед отправкой источнику запроса.

6.1.2. Проверка ошибок

Механизмы проверки ошибок не позволяют недействительным или повторяющимся запросам достичь хостов сервисов, чтобы те обрабатывали только действительные запросы:

- *Проверка запросов*: один шаг проверяет, что запрос правильно отформатирован. Например, тело запроса POST должно содержать действительную разметку JSON. Проверка гарантирует, что в запросе присутствуют все обязательные параметры и их значения соответствуют ограничениям. Эти требования для сервисов можно настроить на шлюзе API.
- *Дедупликация запросов*: дублирование может возникнуть, когда успешный запрос не достигает источника запроса/клиента, поскольку источник запроса/клиент пытается повторить запрос. Кэширование обычно используется для хранения ранее встречавшихся идентификаторов запросов в целях предотвращения дублирования. Если сервис идемпотентен, не имеет состояния или гарантирует, что событие будет доставлено «хотя бы один раз», он обработает дублирующиеся запросы и ошибок не будет. Но если сервис ожидает доставку «ровно один раз» или «не более одного раза», дублирование запросов может породить ошибки.

6.1.3. Производительность и доступность

Шлюз API может повысить производительность и доступность сервисов, предоставляя функциональность кэширования, ограничения частоты и диспетчеризации запросов.

- *Кэширование*: шлюз API может кэшировать частые запросы к базе данных или предоставлять другие возможности, например:
  - В нашей архитектуре сервиса шлюз API может выдавать запросы к сервису метаданных (см. раздел 6.3). Он может кэшировать информацию о самых часто используемых сущностях.

  - Использование идентификационных данных для экономии обращений к сервисам аутентификации и авторизации.
- *Ограничение частоты* (также называемое *регулированием*): предотвращает перегрузку сервиса запросами. (Пример сервиса ограничения частоты представлен в главе 8.)
- *Диспетчеризация запросов*: шлюз API направляет удаленные вызовы к другим сервисам. Он создает клиенты HTTP для этих сервисов и следит, чтобы запросы к этим сервисам были должным образом изолированы. Когда один сервис сталкивается с замедлением, это не влияет на запросы к другим сервисам. Распространенные паттерны, такие как Bulkhead и «Предохранитель» (Circuit breaker), помогают реализовать изоляцию ресурсов и повышают стабильность сервисов при сбое удаленных вызовов.

6.1.4. Ведение журналов и аналитика

Другая распространенная функциональность, предоставляемая шлюзом API, — ведение журналов запросов или сбор данных об использовании ресурсов, то есть сбор информации в реальном времени для различных целей: аналитики, аудита, выставления счетов, отладки и т. д.

6.2. ПАТТЕРН «СЕРВИСНАЯ СЕТЬ» (SERVICE MESH)/SIDECAR

В разделе 1.4.6 мы кратко рассмотрели использование сервисной сети для устранения основных недостатков шлюза API. Кратко напомним эти недостатки:

- увеличение задержки при каждом запросе из-за необходимости маршрутизировать запрос через дополнительный сервис;
- большой кластер хостов, требующий масштабирования для управления затратами.

Рисунок 6.1 повторяет схему сервисной сети, приведенную на рис. 1.8. Небольшой недостаток этого дизайна заключается в том, что хост сервиса будет недоступен в случае недоступности sidecar-контейнера, даже если сервис активен; именно по этой причине мы обычно не запускаем несколько сервисов или контейнеров на одном хосте.

В документации Istio указано, что сервисная сеть состоит из области управления и области данных (https://istio.io/latest/docs/ops/deployment/architecture/), тогда как Дженн Гайл (Jenn Gile) из Nginx включает область наблюдаемости (https://www.nginx.com/blog/how-to-choose-a-service-mesh/). На рис. 6.1 представлены все три типа областей.

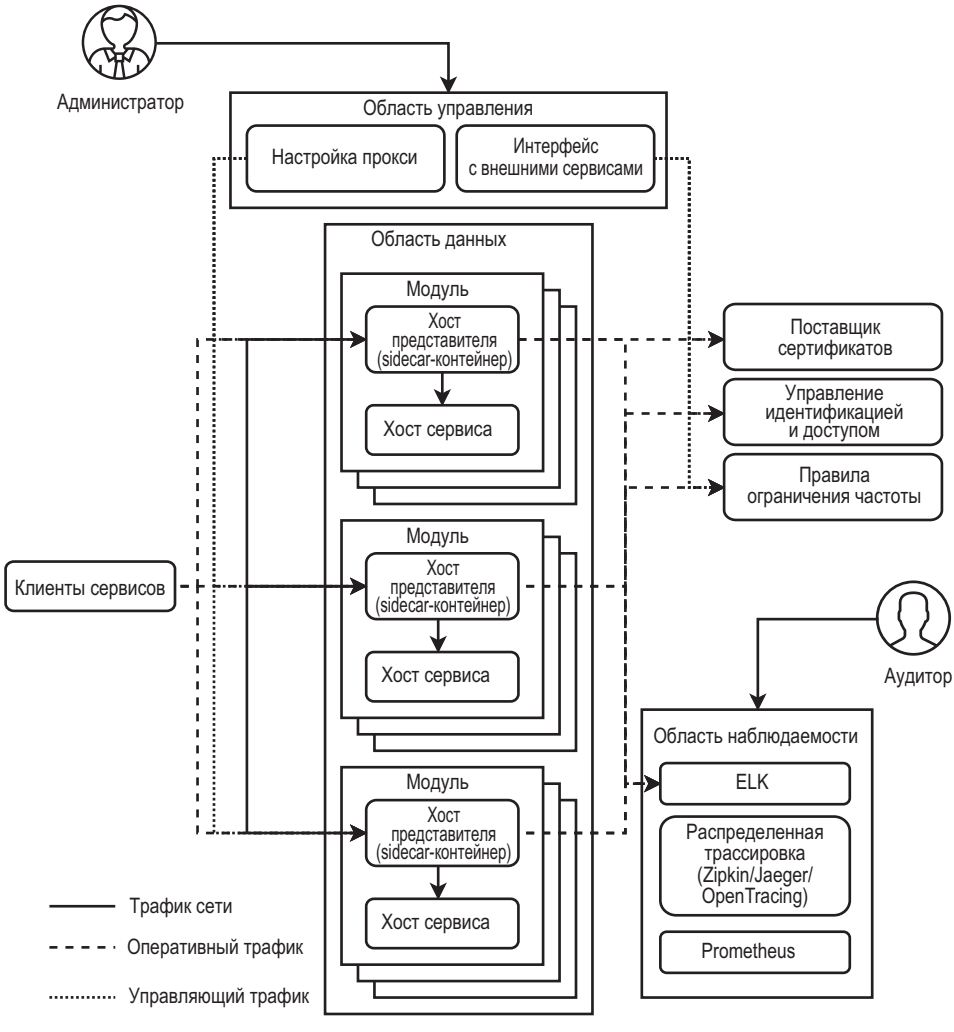

Рис. 6.1. Схема сервисной сети (повторение рис. 1.8)

Администратор может использовать область управления для настройки прокси и взаимодействия с внешними сервисами. Например, область управления может соединяться с поставщиком сертификатов для получения сертификата или с сервисом идентификации и управления доступом для настройки некоторых конфигураций. Она также может передавать идентификатор сертификата или конфигурации сервиса идентификации и управления доступом на прокси-хосты. Между прокси-хостами Envoy (https://www.envoyproxy.io/) передаются межсервисные и внутрисервисные запросы, которые мы будем обозначать термином «трафик сервисной сети». В межсервисных коммуникациях с расширением

могут использоваться различные протоколы, включая HTTP и gRPC (https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/service-mesh-communication-infrastructure). Область наблюдаемости предоставляет функциональность ведения журналов, мониторинга, оповещения и аудита.

Ограничение частоты запросов — еще один пример общего сервиса, который может управляться сервисной сетью. В главе 8 эта тема рассматривается более подробно. AWS App Mesh (https://aws.amazon.com/app-mesh) — облачная сервисная сеть.

ПРИМЕЧАНИЕ Сервисная сеть без расширения кратко рассматривается в разделе 1.4.6.

6.3. СЕРВИС МЕТАДАННЫХ

Сервис метаданных хранит информацию, используемую несколькими компонентами внутри системы. Если компоненты должны передавать эту информацию друг другу, они могут передавать только идентификаторы, а не все данные. Компонент, получивший идентификатор, запрашивает у сервиса метаданных информацию, соответствующую этому идентификатору. Такое решение сокращает дублирование информации в системе по аналогии с нормализацией SQL, так что оно улучшает согласованность данных.

Один из примеров такого рода — пайплайны ETL. Возьмем пайплайн ETL для отправки приветственных сообщений по продуктам, на которые подписались пользователи. Сообщение может представлять собой файл HTML, который содержит несколько мегабайт текста и графики, относящихся к конкретному продукту. На рис. 6.2 при отправке сообщения в очередь пайплайна производитель включает в сообщение не весь файл HTML, а только его идентификатор. Файл хранится в сервисе метаданных. Когда потребитель потребляет сообщение, он запрашивает у сервиса метаданных файл HTML, соответствующий этому идентификатору. Такой подход избавляет от необходимости хранения больших объемов дублирующихся данных в очереди.

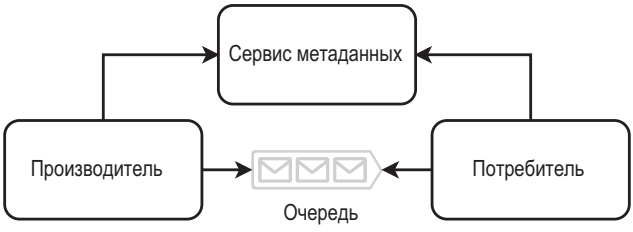

Рис. 6.2. Сервис метаданных может использоваться для сокращения размера отдельных сообщений в очереди за счет хранения больших объектов в сервисе метаданных и размещения в очереди только идентификаторов из сообщений

За использование сервиса метаданных приходится расплачиваться повышением сложности и увеличением общей задержки. Теперь производитель должен выполнять запись как в сервис метаданных, так и в очередь. В некоторых схемах дизайна можно заполнить сервис метаданных на более раннем шаге, так что производителю не нужно выполнять запись в сервис метаданных.

Если в кластере производителя произошел выброс трафика, он будет выдавать запросы на чтение к сервису метаданных с высокой частотой, так что сервис метаданных должен быть способен обеспечить чтение высоких объемов данных.

Подводя итог: сервис метаданных предназначен для поиска по идентификатору. Сервисы метаданных будут использоваться во многих примерах в части 2.

На рис. 6.3 представлены изменения архитектуры при добавлении шлюза API и сервисов метаданных. Вместо того чтобы отправлять запросы на бэкенд, клиенты направляют их к шлюзу API, который выполняет определенные функции и отправляет запросы сервису метаданных и/или бэкенду. Сервисная сеть изображена на рис. 1.8.

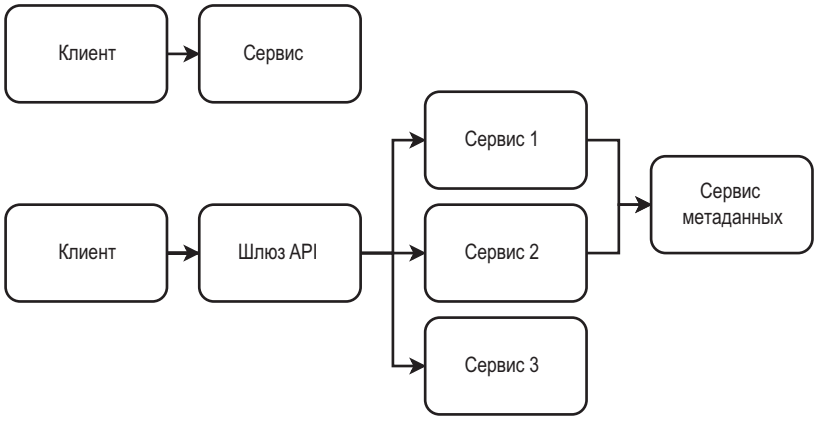

Рис. 6.3. Функциональное секционирование сервиса (сверху) для выделения шлюза API и сервиса метаданных (снизу). До секционирования клиенты обращаются с запросами к сервису напрямую. После секционирования клиенты обращаются с запросами к шлюзу API, который выполняет определенные функции и отправляет запросы одному из сервисов, который, в свою очередь, запрашивает у сервиса метаданных определенные общие виды функциональности

6.4. ОБНАРУЖЕНИЕ СЕРВИСОВ

Обнаружение сервисов (service discovery) — концепция микросервисов, которую можно кратко упомянуть на собеседовании в контексте управления несколькими сервисами. Обнаружение сервисов выполняется «под капотом», и большинству инженеров не нужно знать его подробности. Обычно достаточно понимать,

что каждому внутреннему сервису API назначается номер порта, по которому к нему поступают обращения. Внешним сервисам API и многим UI-сервисам назначаются URL для обращений. Обнаружение сервисов может обсуждаться на собеседованиях для команд, занимающихся разработкой инфраструктуры. Вряд ли о нем пойдет речь в случае отбора кандидатов на другие позиции, поскольку оно не даст полезной информации.

Очень кратко: обнаружение сервисов — механизм, при помощи которого клиенты получают информацию о том, какие хосты сервисов им доступны. Реестр сервисов — база данных, в которой хранится информация о доступных хостах сервиса. За дополнительной информацией о реестрах сервисов в Kubernetes и AWS обращайтесь к таким источникам, как https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/service-discovery.html. Подробно об обнаружении на стороне клиента и на стороне сервера рассказывается здесь: https://microservices.io/patterns/client-side-discovery.html и https://microservices.io/patterns/server-side-discovery.html.

6.5. ФУНКЦИОНАЛЬНОЕ СЕКЦИОНИРОВАНИЕ И РАЗЛИЧНЫЕ ФРЕЙМВОРКИ

В этом разделе описывается только небольшая часть из множества фреймворков, которые можно использовать для разработки компонентов систем. В отрасли постоянно появляются новые фреймворки, одни становятся популярными, а другие выходят из моды. Их количество может вызвать замешательство у новичка. Более того, некоторые фреймворки могут использоваться для нескольких компонентов, что еще более усложняет общую схему системы. В этом разделе приведено общее описание фреймворков, таких как:

- веб-платформы;
- мобильные платформы, включая Android и iOS;
- бэкенд;
- десктопные.

Вселенная языков и фреймворков намного шире, чем можно описать в этом разделе, и мы не будем пытаться охватить ее целиком. Цель раздела — дать вам представление о некоторых фреймворках и языках. К концу его вам будет проще читать документацию фреймворков, чтобы понять их предназначение и место в проектировании систем.

6.5.1. Базовый дизайн системы приложения

На рис. 1.1 (в главе 1) изображен базовый дизайн системы приложения. В наши дни компании, которые разрабатывают мобильные приложения, обращающиеся

с запросами к бэкенд-сервису, почти всегда размещают приложения для iOS в магазине приложений iOS, а приложения для Android в магазине Google Play. Они также могут разработать браузерное приложение с такими же возможностями, как у мобильной версии, и возможно, простую страницу со ссылками для загрузки мобильных приложений. Вариантов много, в том числе и десктопное приложение. Однако объяснять все возможные комбинации нецелесообразно, и мы этим заниматься не станем.

Начнем с рассмотрения вопросов, относящихся к рис. 1.1, а затем расширим обсуждение до разных фреймворков и их языков:

- Почему приложение веб-сервера существует отдельно от бэкенда и браузерного приложения?
- Почему браузерное приложение отправляет запросы к приложению Node.js, которое затем отправляет запросы к бэкенду, общему для Android- и iOS-приложений?

6.5.2. Назначение приложения веб-сервера

Приложение веб-сервера служит для реализации следующих целей:

- Когда клиент, использующий веб-браузер, обращается к некоторому URL (например, https://google.com/), браузер загружает приложение из приложения Node.js. Как упоминалось в разделе 1.4.1, браузерное приложение должно быть по возможности небольшим, чтобы его загрузка не занимала много времени.
- Когда браузер направляет запрос к конкретному URL (например, с путем вида https://google.com/about), Node.js выполняет маршрутизацию URL и предоставляет соответствующую страницу.
- URL может включать пути и параметры запросов, которые требуют определенных запросов к бэкенду. Приложение Node.js обрабатывает URL и направляет соответствующие запросы к бэкенду.
- Некоторые действия пользователя в браузерном приложении (такие, как заполнение и отправка данных форм или щелчки на кнопках) могут требовать запросов к бэкенду. Одно действие может соответствовать нескольким запросам к бэкенду, так что приложение Node.js предоставляет собственный API браузерному приложению. На рис. 6.4 для каждого действия пользователя браузерное приложение направляет запрос API к серверу/приложению Node.js, которое затем направляет один или несколько запросов к бэкенду и возвращает запрошенные данные.

Почему браузер не направляет запросы сразу к бэкенду? Если бэкенд является REST-приложением, его конечные точки не возвращают именно те данные, которые нужны браузеру. Возможно, браузеру придется направить несколько

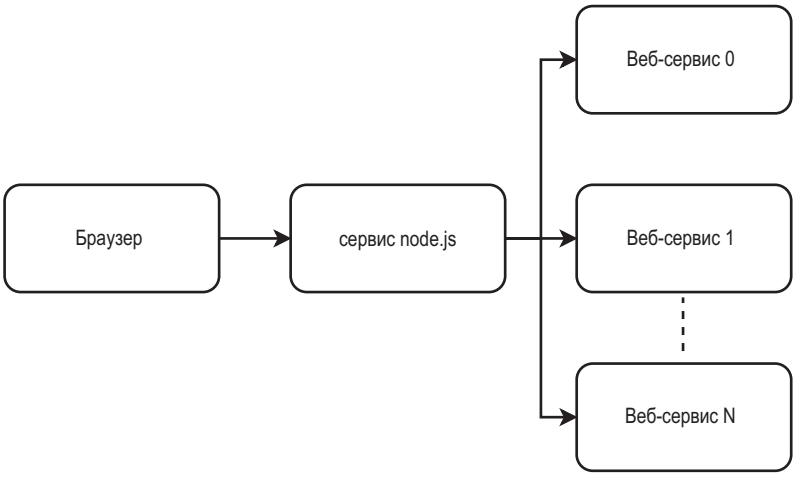

Рис. 6.4. Сервер Node.js обслуживает запрос от браузера: он направляет соответствующие запросы к одному или нескольким веб-сервисам, агрегирует и обрабатывает их ответы и возвращает итоговый ответ браузеру

запросов API и получить больше данных, чем необходимо. Данные передаются по интернету между устройством пользователя и датацентром, что неэффективно. Эффективнее направлять эти большие запросы из приложения Node.js, потому что передача данных, скорее всего, будет происходить между соседними хостами одного датацентра. Тогда приложение Node.js вернет именно те данные, которые нужны браузеру.

Приложения, использующие GraphQL, позволяют точно запросить необходимые данные, но защита конечных точек GraphQL сложнее, чем в REST; ее обеспечение чревато увеличением времени разработки и дефектами безопасности. Существуют и другие недостатки, перечисленные ниже (дополнительную информацию о GraphQL см. в разделе 6.7.4):

- больше усилий по оптимизации производительности из-за гибкости запросов;
- увеличение объема кода на стороне клиента;
- дополнительные усилия по определению схемы;
- увеличение размера запросов.

6.5.3. Веб- и мобильные фреймворки

В этом разделе перечислены фреймворки следующих категорий:

- для разработки веб-/браузерных приложений;
- для разработки мобильных приложений;

- для разработки бэкенд-приложений;
- для разработки десктопных приложений (Windows, Mac, Linux).

Полный список получится очень длинным, и в него войдут фреймворки, с которыми вы вряд ли когда-либо будете работать (и даже вряд ли прочитаете о них). По нашему мнению, составлять такой список бессмысленно. Поэтому мы перечислим только те фреймворки, которые заслуживают внимания (или когда-то заслуживали его).

Гибкость областей применения фреймворков усложняет их полное и объективное обсуждение. Фреймворки и языки развиваются в самых разных направлениях; где-то логично, а где-то — нет.

Разработка браузерных приложений

Браузеры принимают только HTML, CSS и JavaScript, поэтому в целях обеспечения обратной совместимости браузерные приложения должны быть написаны на этих языках. Браузер устанавливается на устройстве пользователя; следовательно, он должен обновляться самим пользователем, и слишком сложно и непрактично убеждать или заставлять пользователей загружать браузер, поддерживающий новый язык. Браузерное приложение можно написать на ванильном JavaScript (без использования фреймворков), но этот вариант подойдет разве что для самых маленьких браузерных приложений, потому что фреймворки содержат много функций, которые пришлось бы заново реализовать на ванильном JavaScript (например, анимации, сортировка таблиц или построение диаграмм).

Хотя браузерные приложения должны быть написаны на этих трех языках, фреймворк может поддерживать другие языки. Код браузерного приложения, написанный на этих языках, транспилируется на HTML, CSS и JavaScript.

Самые популярные фреймворки браузерных приложений — React, Vue.js и Angular. Среди других фреймворков стоит упомянуть Meteor, jQuery, Ember.js и Backbone.js. У всех этих фреймворков есть одна особенность: разработчики смешивают разметку и логику в одном файле, вместо того чтобы использовать файлы HTML для разметки, а файлы JavaScript для логики. Эти фреймворки также могут содержать собственные языки для разметки и логики. Например, в React был введен JSX — язык разметки, похожий на HTML. Файл JSX может включать как разметку, так и функции и классы JavaScript. В Vue.js поддерживается шаблонный тег, сходный с HTML.

Некоторые наиболее заметные языки веб-разработки (транспилируемые в JavaScript):

- TypeScript (https://www.typescriptlang.org/) — язык со статической типизацией. Представляет собой обертку/надмножество JavaScript. TypeScript может использоваться практически всеми фреймворками JavaScript, хотя это потребует некоторой предварительной настройки.

- Elm (https://elm-lang.org/) может напрямую транспилироваться в HTML, CSS и JavaScript или использоваться с другими фреймворками, такими как React.
- PureScript (https://www.purescript.org/) предоставляет синтаксис, сходный с синтаксисом Haskell.
- Reason (https://reasonml.github.io/).
- ReScript (https://rescript-lang.org/).
- Clojure (https://clojure.org/) — язык общего назначения. Фреймворк ClojureScript (https://clojurescript.org/) транспилирует Clojure в JavaScript.
- CoffeeScript (https://coffeescript.org/).

Эти фреймворки браузерных приложений предназначены для стороны браузера/клиента. Также существуют фреймворки для стороны сервера. Каждый фреймворк на стороне сервера также может выдавать запросы к базам данных и использоваться для разработки бэкенда. На практике компания часто выбирает один фреймворк для серверной разработки и другой для разработки бэкенда. Это часто сбивает с толку новичков, пытающихся разделять фреймворки «фронтенда на стороне сервера» и фреймворки «бэкенда». Между ними нет четкой границы.

- Express (https://expressjs.com/) — серверный фреймворк для Node.js (https://nodejs.org/). Node.js — исполнительная среда JavaScript на основе ядра Chrome V8 JavaScript. Ядро V8 JavaScript изначально было построено для Chrome, но оно может работать и в операционной системе (например, Linux или Windows). Среда Node.js обеспечивает выполнение кода JavaScript в операционной системе. Многие объявления о вакансиях, в которых Node.js указывается как требование для фронтенд- или полностековой разработки, на самом деле подразумевают знание Express.
- Deno (https://deno.land/) поддерживает JavaScript и TypeScript. Райан Даль (Ryan Dahl), создатель оригинальной версии Node.js, разработал этот фреймворк, разочаровавшись в Node.js.
- Goji (https://goji.io/) — фреймворк для Golang.
- Rocket (https://rocket.rs/) — фреймворк для Rust. Примеры использования веб-сервиса Rust и фреймворков бэкенда см. на странице https://blog.logrocket.com/the-current-state-of-rust-web-frameworks/.
- Vapor (https://vapor.codes/) — фреймворк для языка Swift.
- Vert.x (https://vertx.io/) предоставляет средства разработки на Java, Groovy и Kotlin.
- PHP (https://www.php.net/). (Единого мнения о том, что собой представляет PHP — язык или фреймворк, — до сих пор не существует. Мы считаем, что эти споры не имеют практической ценности.) Популярный технологический стек обозначается аббревиатурой LAMP (Linux, Apache, MySQL, PHP/

Perl/Python). Код PHP может выполняться на сервере Apache (https://httpd.apache.org/), который, в свою очередь, выполняется на хосте Linux. Пик популярности PHP наблюдался примерно до 2010 года (https://www.tiobe.com/tiobe-index/php/), но по нашему опыту, в новых проектах чистый код PHP используется редко. PHP занимает видное место в веб-разработке на базе платформы WordPress, предназначенной для создания простых веб-сайтов. Более сложные пользовательские интерфейсы и настройки проще создавать с применением фреймворков, требующих основательного программирования, таких как React и Vue.js. Платформа Meta (ранее Facebook) широко использовала PHP. Браузерное приложение Facebook было разработано на PHP. В 2014 году компания Facebook представила язык Hack (https://hacklang.org/) и виртуальную машину HipHop (HHVM). Hack — язык, похожий на PHP, но не имеющий недостатков PHP в отношении безопасности и производительности. Он работает на машине HVVM. Hack и HHVM широко используются компанией Meta.

Разработка мобильных приложений

В области мобильных операционных систем доминируют Android и iOS, разработанные компаниями Google и Apple соответственно. Как Google, так и Apple предлагают собственные платформы разработки приложений Android или iOS, обычно называемые нативными (native) платформами. Нативными языками разработки Android являются Kotlin и Java, а нативными языками разработки iOS — Swift и Objective-C.

Кросс-платформенная разработка

Теоретически фреймворки кросс-платформенной разработки снижают степень дублирования работы за счет выполнения одного кода на разных платформах. На практике для реализации приложения на каждой платформе обычно приходится писать дополнительный код, который нивелирует часть этого преимущества. Так бывает, когда компоненты пользовательского интерфейса (UI), предоставляемые операционными системами, сильно отличаются друг от друга. Примеры фреймворков, считающихся кросс-платформенными между Android и iOS:

- React Native (не путайте с React, последний предназначен только для веб-разработки). Также существует фреймворк React Native for Web (https://github.com/necolas/react-native-web), который позволяет выполнять веб-разработку с использованием React Native.
- Flutter (https://flutter.dev/) — кросс-платформенный фреймворк для Android, iOS, веб- и десктопной разработки.
- Ionic (https://ionicframework.com/) — кросс-платформенный фреймворк для Android, iOS, веб- и десктопной разработки.

- Xamarin (https://dotnet.microsoft.com/en-us/apps/xamarin) — кросс-платформенный фреймворк для Android, iOS и Windows.
- Electron (https://www.electronjs.org/) — кросс-платформенный фреймворк для веб- и десктопной разработки.
- Cordova (https://cordova.apache.org/) — фреймворк для мобильной и десктопной разработки с использованием HTML, CSS и JavaScript. Cordova делает возможной кросс-платформенную разработку с такими фреймворками веб-разработки, как Ember.js.

Другой метод основан на программировании *прогрессивных веб-приложений*, или PWA (Progressive Web App). PWA — браузерное или веб-приложение, обладающее всеми стандартными возможностями настольного браузера и использующее некоторые функции (такие, как скрипты *service workers* и *манифесты приложений*) для обеспечения пользовательского опыта, аналогичного работе с мобильными устройствами. Например, при помощи скриптов service workers прогрессивное веб-приложение может предоставлять такие возможности, как активные оповещения и кэширование данных в браузере, что позволяет работать с ним как с нативным мобильным приложением. Разработчик может настроить манифест приложения, чтобы PWA-приложение можно было установить на настольном компьютере или мобильном устройстве. Пользователь может добавить значок приложения на домашний экран своего устройства, в меню запуска или на рабочий стол и открыть приложение касанием этого значка; процесс напоминает установку приложений из магазинов Android или iOS. Так как размеры экранов разных устройств разные, разработчики и проектировщики должны использовать *отзывчивый дизайн веб-приложений*; при таком подходе веб-приложение хорошо работает на экранах разных размеров или при изменении размера окна браузера. Разработчики могут применять такие решения, как *медийные запросы* (https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries) или `ResizeObserver` (https://developer.mozilla.org/en-US/docs/Web/API/ResizeObserver), чтобы приложение рендерилось под меняющиеся размеры окна браузера или экрана.

Бэкенд-разработка

Ниже перечислены фреймворки бэкенд-разработки. Фреймворки бэкенда можно разделить на три группы, реализующие технологии RPC, REST и GraphQL. Некоторые фреймворки бэкенд-разработки являются полностековыми, то есть могут использоваться для разработки монолитного браузерного приложения, обращающегося с запросами к базе данных. Также они могут использоваться для разработки браузерных приложений и отправки запросов к сервису бэкенда, разработанному в другом фреймворке, но мы никогда не встречали случаев такого применения:

- gRPC (https://grpc.io/) — фреймворк RPC, который может использоваться для разработки на C#, C++, Dart, Golang, Java, Kotlin, Node, Objective-C, PHP,

Python или Ruby. В будущем также может быть расширен для поддержки других языков.

- Thrift (https://thrift.apache.org/) и Protocol Buffers (https://developers.google.com/protocol-buffers) используются для сериализации объектов данных, сжимая их для сокращения сетевого трафика. Объект определяется в файле определения. Тогда код клиента и сервера (бэкенда, не веб-сервера) генерируется по файлу определения. Клиенты используют код клиента для сериализации запросов к бэкенду, на котором используется код бэкенда для десериализации запросов, и наоборот — для ответов от бэкенда. Файлы определений также помогают поддерживать обратную и прямую совместимость за счет установки ограничений на возможные изменения.
- Dropwizard (https://www.dropwizard.io/) — пример REST-фреймворка Java. Spring Boot (https://spring.io/projects/spring-boot) может использоваться для создания приложений Java, включая REST-сервисы.
- Flask (https://flask.palletsprojects.com/) и Django (https://www.djangoproject.com/) — два примера REST-фреймворков Python. Они также могут использоваться для разработки веб-серверов.

Несколько примеров полностековых фреймворков:

- Dart (https://dart.dev) — язык, предоставляющий фреймворки для любого решения. Может использоваться для полностековой разработки, бэкенд-разработки, серверной, браузерной разработки и мобильных приложений.
- Rails (https://rubyonrails.org/) — полностековый фреймворк для Ruby, который также можно использовать для REST. Ruby on Rails часто используется как единое решение (вместо использования Ruby с другими фреймворками или Rails с другими языками).
- Yesod (https://www.yesodweb.com/) — фреймворк для Haskell, который также можно использовать для REST. Разработка браузерных приложений может осуществляться на Yesod с использованием языка шаблонов Shakespearean (https://www.yesodweb.com/book/shakespearean-templates), который транспилируется в HTML, CSS и JavaScript.
- Integrated Haskell Platform (https://ihp.digitallyinduced.com/) — еще один фреймворк для Haskell.
- Phoenix (https://www.phoenixframework.org/) — фреймворк для языка Elixir.
- JavaFX (https://openjfx.io/) — платформа клиентских приложений Java для настольных, мобильных и встроенных систем. Происходит от фреймворка Java Swing (https://docs.oracle.com/javase/tutorial/uiswing/), предназначенного для разработки GUI для Java-программ.
- Beego (https://beego.vip/) и Gin (https://gin-gonic.com/) — фреймворки для Golang.

6.6. БИБЛИОТЕКИ И СЕРВИСЫ

После определения компонентов системы можно обсудить достоинства и недостатки реализации каждого компонента в виде библиотеки или сервиса на стороне клиента либо на стороне сервера. Не торопитесь, если думаете, что для каждого конкретного компонента лучше подходит тот или иной вариант. В большинстве ситуаций выбор между использованием библиотеки и сервиса не очевиден, поэтому вы должны уметь обсуждать дизайн, подробности реализации и компромиссы для обоих вариантов.

Библиотека может иметь вид независимого программного пакета, тонкой прослойки, которая направляет все запросы и ответы между клиентами и серверами соответственно, или содержать элементы обоих решений. Другими словами, некоторая часть логики API реализуется внутри библиотеки, а остальная — сервисами, вызываемыми библиотекой. В этой главе в контексте сравнения библиотек с сервисами термином «библиотека» будет обозначаться независимая библиотека.

В табл. 6.1 представлено сравнение библиотек с сервисами. Многие из этих критериев более подробно рассматриваются в оставшейся части главы.

Таблица 6.1. Сравнение библиотек с сервисами

Библиотека	Сервис
Пользователи выбирают, какую версию/сборку использовать, и имеют больше свободы при обновлении новых версий	Разработчики выбирают сборку и управляют тем, когда происходит обновление
Недостаток в том, что пользователи могут пользоваться старыми версиями библиотек, содержащими ошибки или дефекты безопасности, исправленные в новых версиях	
Пользователи, желающие всегда использовать последнюю версию часто обновляемой библиотеки, должны реализовать программные обновления самостоятельно	
Отсутствие взаимодействий или обмена данными между устройствами ограничивает приложения. Если пользователем является другой сервис, этот сервис масштабируется горизонтально и между хостами должен происходить обмен данными, хосты клиентского сервиса должны иметь возможность взаимодействовать друг с другом для обмена данными. Этот обмен данными должен быть реализован разработчиками пользовательского сервиса	Ограничение отсутствует. Синхронизация данных между хостами может осуществляться путем их запросов друг к другу или к базе данных. Пользователям об этом можно не беспокоиться

Таблица 6.1 (*окончание*)

Библиотека	Сервис
Привязка к языку	Технологическая нейтральность
Предсказуемая задержка	Менее предсказуемая задержка из-за зависимости от состояния сетевого подключения
Предсказуемое, воспроизводимое поведение	Сетевые проблемы непредсказуемы и плохо воспроизводимы, так что поведение может быть менее предсказуемым и менее воспроизводимым
Если понадобится масштабировать нагрузку на библиотеку, все приложение должно масштабироваться вместе с ней. Затраты на масштабирование несет сервис пользователя	Масштабируется независимо. Затраты на масштабирование несет сервис
У пользователя есть возможность декомпилировать код в целях хищения интеллектуальной собственности	Пользователь не имеет доступа к коду. (Хотя реверсивный инжиниринг API возможен, но эта тема выходит за рамки книги)

6.6.1. Привязка к языку и технологическая нейтральность

Для простоты использования библиотека должна быть написана на языке клиента, так что ее необходимо реализовать на каждом языке, который она поддерживает.

Большинство библиотек оптимизировано для выполнения четко определенного набора взаимосвязанных задач, что позволяет реализовать их на одном языке. Однако некоторые библиотеки могут быть частично или полностью написаны на другом языке, поскольку определенные языки и фреймворки лучше подходят для конкретных целей. Реализация всей логики на одном языке в таком случае может привести к неэффективному использованию библиотеки. Более того, в процессе разработки библиотеки иногда возникает необходимость в использовании библиотек, написанных на других языках. Существуют вспомогательные библиотеки, которые могут использоваться для разработки библиотек, содержащих компоненты на других языках. Эта тема выходит за рамки книги. Практическая трудность такого подхода в том, что команде или компании, разрабатывающей библиотеку, потребуются инженеры, свободно владеющие всеми этими языками.

Сервис, в отличие от библиотеки, является технологически нейтральным, поскольку клиент может использовать сервис независимо от его текущего или прошлого технологического стека. Сервис может быть реализован на языке и во фреймворках, наиболее подходящих для его целей. При этом появляются

небольшие накладные расходы для клиентов, которым приходится создавать и поддерживать подключения HTTP, RPC или GraphQL к сервису.

6.6.2. Предсказуемость задержки

Библиотека не имеет сетевой задержки, обеспечивает гарантированное и предсказуемое время ответа и может легко профилироваться такими инструментами, как флейм-графики (flame graph).

Сервис имеет непредсказуемую и неконтролируемую задержку, так как она зависит от множества факторов, в числе которых:

- сетевая задержка, зависящая от качества подключения пользователя к интернету;
- способность сервиса обрабатывать текущий объем трафика.

6.6.3. Предсказуемость и воспроизводимость поведения

Поведение сервиса менее предсказуемо и хуже воспроизводимо, чем поведение библиотеки, поскольку оно имеет больше зависимостей, включая следующие.

- Развертывание обычно происходит постепенно (то есть сборка развертывается на части хостов сервиса за раз). Балансировщик нагрузки маршрутизирует запросы хостам, на которых работают разные сборки, что приведет к различиям в поведении.
- Пользователи не имеют полного контроля над данными сервиса, и разработчики сервиса могут менять эти данные в перерывах между запросами. В этом сервис отличается от библиотек, в которых пользователи полностью контролируют файловую систему машины.
- Сервис может отправлять запросы к другим сервисам, и непредсказуемое и невоспроизводимое поведение этих сервисов отразится на нем.

Несмотря на эти факторы, сервис часто проще отлаживать, чем библиотеку:

- Разработчики сервиса имеют полный доступ к его журналам, тогда как разработчики библиотеки не имеют полного доступа к журналам на устройствах пользователей.
- Разработчики сервиса управляют исполнительной средой и могут настроить унифицированную среду с использованием таких средств, как виртуальные машины и Docker для хостов. Библиотека может запускаться пользователями в разнообразных средах с разным оборудованием, прошивками и ОС (Android/iOS). Пользователь может отправлять протоколы сбоя разработчикам, но без доступа к устройству пользователя и точной конфигурации среды отладку проводить сложнее.

6.6.4. Особенности масштабирования библиотек

Библиотека не может масштабироваться независимо, поскольку она содержится в приложении пользователя. Нет смысла обсуждать масштабирование библиотеки на одном пользовательском устройстве. Если приложение пользователя работает параллельно на нескольких устройствах, пользователь может масштабировать библиотеку посредством масштабирования использующего ее приложения. Чтобы масштабировать только библиотеку, пользователь может создать собственный сервис-обертку для библиотеки и масштабировать этот сервис. Но тогда это будет уже не библиотека, а просто сервис, принадлежащий пользователю, так что затраты на масштабирование будет нести пользователь.

6.6.5. Другие факторы

В этом разделе мы приведем несколько практических наблюдений из нашего личного опыта.

Некоторые инженеры неохотно упаковывают свой код в библиотеки, но спокойно относятся к соединению с сервисами. Их смущает то, что библиотека увеличит размер сборки, особенно для пакетов JavaScript. Их также беспокоит возможность появления вредоносного кода в библиотеках, тогда как с сервисами такой проблемы нет, ведь инженеры контролируют данные, отправляемые сервисам, и им полностью видны ответы сервиса.

Изменения, нарушающие обратную совместимость, в библиотеках, в отличие от сервисов, особенно внутренних, ожидаемы. Разработчикам сервисов иногда приходится соблюдать нелепые соглашения об именах конечных точек API, например, добавлять к ним суффиксы вида «/v2», «/v3» и т. д.

Опыт подсказывает, что паттерн «Адаптер» чаще применяется при использовании библиотек, а не сервисов.

6.7. РАСПРОСТРАНЕННЫЕ ПАРАДИГМЫ API

В этом разделе приводится краткое описание и сравнение основных коммуникационных парадигм. При выборе парадигмы для сервиса необходимо учитывать их сильные и слабые стороны:

- REST (Representational State Transfer);
- RPC (Remote Procedure Call);
- GraphQL;
- WebSocket.

6.7.1. Модель OSI (Open Systems Interconnection)

Семиуровневая модель OSI — концептуальная модель/фреймворк, характеризующая функции сетевой системы без учета ее внутренней структуры и технологии. В табл. 6.2 кратко описаны все уровни. Эта модель удобна с той точки зрения, что протоколы каждого уровня реализуются в ней с использованием протоколов нижнего уровня.

Actor, GraphQL, REST и WebSocket реализованы на основе HTTP. RPC классифицируется как протокол 5-го уровня, поскольку он работает с подключениями, портами и сеансами напрямую, а не полагается на протокол более высокого уровня (такой, как HTTP).

Таблица 6.2. Модель OSI

Номер уровня	Название	Описание	Примеры
7	Прикладной уровень	Пользовательский интерфейс	FTP, HTTP, Telnet
6	Уровень представления	Обеспечивает представление данных. На этом уровне выполняется шифрование	UTF, ASCII, JPEG, MPEG, TIFF
5	Сеансовый уровень	Различает данные разных приложений. Поддерживает подключения. Управляет портами и сеансами	RPC, SQL, NFX, X Windows
4	Транспортный уровень	Сквозные подключения. Определяет надежную или ненадежную доставку и логику управления потоком	TCP, UDP
3	Сетевой уровень	Логическая адресация. Определяет физический путь для данных. На этом уровне работают маршрутизаторы	IP, ICMP
2	Канальный уровень	Сетевой формат. Может исправлять ошибки физического уровня	Ethernet, wi-i
1	Физический уровень	Обычные биты на физическом носителе	Оптоволокно, коаксиальный кабель, повторитель, модем, сетевой адаптер, USB

6.7.2. REST

Предполагается, что читатель знаком с основами REST как коммуникационной архитектуры без состояния, использующей методы HTTP и тело запроса/ответа, чаще всего кодируемое в JSON или XML. В этой книге мы используем REST для API и JSON для запросов POST и тела ответа. Схему JSON можно представить в виде JSON Schema (https://json-schema.org/), но в книге мы этого делать не будем, потому что обычно подробное обсуждение схем JSON получается слишком пространным и низкоуровневым, чтобы включать его в 50-минутное собеседование по проектированию систем.

Архитектура REST проста в изучении, подготовке, экспериментах и отладке (с использованием curl или клиента REST). Среди других преимуществ можно отметить возможности гипермедиа и средства кэширования, которые будут рассмотрены ниже.

Гипермедиа

Суть гипермедиа-элементов управления (HATEOAS), или гипермедиа, заключается в предоставлении клиенту информации о «следующих доступных действиях» в ответе. Гипермедиа реализуется в форме полей, например «ссылок» с JSON-разметкой ответа, содержащей конечные точки API, к которым клиенту было бы логично обратиться с запросом на следующем шаге.

Например, после того как интернет-магазин выведет на экран счет, следующим шагом для клиента становится оплата. Тело ответа для конечной точки счета может содержать ссылку на конечную точку платежа:

```
{
  "data": {
    "type": "invoice",
    "id": "abc123",
  },
  "links": {
    "pay": "https://api.acme.com/payment/abc123"
  }
}
```

где ответ содержит идентификатор счета, а следующим шагом будет отправка POST-запроса платежа для этого идентификатора счета.

Также имеется метод HTTP OPTIONS, предназначенный для извлечения метаданных о конечной точке: доступных действий, полей, которые могут обновляться, или данных, которые ожидаются в определенных полях.

На практике гипермедиа и OPTIONS сложны для разработчиков клиентской части, и разумнее предоставить разработчику клиента документацию API для каждой конечной точки или функции, например OpenAPI (https://swagger.io/speciication/) для REST, или встроенную документацию для фреймворков RPC

и GraphQL. Текст соглашений о спецификации тела JSON запросов/ответов находится по адресу https://jsonapi.org/.

Другие коммуникационные архитектуры, такие как RPC или GraphQL, не поддерживают гипермедиа.

Кэширование

Разработчикам следует объявлять ресурсы REST кэшируемыми там, где это возможно; это даст целый ряд преимуществ:

- снижение задержки, так как некоторых сетевых вызовов можно будет избежать;
- повышение доступности, потому что ресурс доступен, даже если недоступен сервис;
- улучшение масштабируемости вследствие снижения нагрузки на сервер.

Для кэширования используйте заголовки HTTP `Expires`, `Cache-Control`, `ETag` и `Last-Modiied`.

Заголовок HTTP `Expires` задает абсолютный срок жизни для кэшированного ресурса. Можно задать время вплоть до одного года от текущих показаний часов. Пример заголовка: `Expires: Mon, 11 Dec 2021 18:00 PST`.

Заголовок `Cache-Control` состоит из директив, разделенных запятыми (инструкций), для кэширования как запросов, так и ответов. Пример заголовка `Cache-Control: max-age=3600` означает, что ответ кэшируется в течение 3600 секунд. Запрос POST или PUT может включать заголовок `Cache-Control` как директиву серверу кэшировать эти данные, но это не означает, что сервер будет следовать такой директиве, и эта директива может не содержаться в ответах запрашиваемых данных. Все директивы кэширования в запросах и ответах описаны на странице https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control.

Значение `ETag` представляет собой непрозрачный строковый токен, который используется в качестве идентификатора конкретной версии ресурса. («Непрозрачным» называется токен, имеющий проприетарный формат, известный только выдающей стороне. Для проверки непрозрачного токена получатель должен обратиться с вызовом к серверу, выдавшему этот токен.) Клиент может более эффективно обновлять свой ресурс, включая значение `ETag` в запрос GET. Сервер возвращает значение ресурса только в том случае, если его значение `ETag` отлично от заданного. Другими словами, значение ресурса изменилось, поэтому сервис не будет без необходимости возвращать значение ресурса, если оно уже имеется у клиента.

Заголовок `Last-Modiied` содержит дату и время последнего изменения ресурса и может использоваться как резервное значение для заголовка `ETag`, если

последний недоступен. С ним связаны заголовки `If-Modiied-Since` и `If-Unmodiied-Since`.

Недостатки REST

Недостатком REST является отсутствие интегрированного механизма документирования, кроме конечных точек гипермедиа или OPTIONS, которые разработчик предоставлять не обязан. К сервису, реализованному с использованием фреймворка REST, необходимо добавить фреймворк документации OpenAPI. В противном случае клиенты не смогут получить информацию о доступных конечных точках запросов, о путях и параметрах запросов или о полях тела запросов и ответов. В REST также нет стандартизированной процедуры версионирования; по принятому соглашению для него используются пути вида «/v2», «/v3» и т. д. Другой недостаток архитектуры REST заключается в том, что у нее нет единой спецификации, что приводит к путанице. Две популярные спецификации REST — OData и JSON-API.

6.7.3. RPC (Remote Procedure Call)

RPC — метод выполнения процедур в другом адресном пространстве (например, на другом хосте), при этом разработчику не важны детали передачи по сети. Среди популярных фреймворков RPC с открытым исходным кодом стоит упомянуть Google gRPC, Facebook Thrift и RPyC для Python.

Вы должны знать следующие распространенные форматы кодирования. Необходимо понимать, как выполняется кодирование (также называемое сериализацией или маршалингом) и декодирование (также называемое парсингом, десериализацией или демаршалингом):

- CSV, XML, JSON;
- Thrift;
- Protocol Buffers (protobuf);
- Avro.

Основные преимущества фреймворков RPC (таких, как gRPC) перед REST:

- Механизм RPC проектировался для оптимизации ресурсов, поэтому эта коммуникационная архитектура лучше всего подходит для маломощных устройств, например устройств IoT для «умных домов». Для больших вебсервисов пониженное потребление ресурсов по сравнению с REST или GraphQL начинает играть важную роль при масштабировании.
- Protobuf — эффективный механизм кодирования. Разметка JSON повторяющаяся и включает много символов, что приводит к увеличению объема запросов и ответов. Экономия сетевого трафика становится более значительной при масштабировании.

- Разработчики определяют схемы своих конечных точек в файлах. Самые популярные форматы — Avro, Thrift и protobuf. Клиенты используют эти файлы для создания запросов и интерпретации ответов. Так как документирование схемы при разработке API обязательно, у разработчиков клиентов всегда хорошая документация API. Эти форматы кодирования также поддерживают правила модификации схем, по которым разработчики понимают, как обеспечить обратную и/или прямую совместимость при модификации.

Главные недостатки механизма RPC следуют из его природы двоичного протокола. Необходимость обновлений файлов схемы до последней версии создает проблемы для клиентов, особенно за пределами организаций. Кроме того, если организация желает отслеживать свой внутренний сетевой трафик, это проще делать с текстовыми протоколами (такими, как REST), чем с двоичными (как RPC).

6.7.4. GraphQL

GraphQL — язык запросов, который делает возможной *декларативную выборку данных*, когда клиент может точно указать, какие данные ему нужны от API. Этот инструмент предоставляет API запросов данных и язык манипулирования для точечных запросов. Он также предоставляет интегрированные средства документирования API, необходимые для достижения подобной гибкости. Главные преимущества GraphQL:

- Клиент решает, какие данные ему нужны и в каком формате.
- Сервер эффективно работает и доставляет ровно то, что запросил клиент, не возвращая данные частично (что требует множественных запросов) или в избыточном количестве (что увеличивает размер ответа).

Минусы:

- Может быть слишком сложным для простых API.
- Освоить его сложнее, чем RPC и REST, включая механизмы безопасности.
- Сообщество пользователей меньше, чем у RPC и REST.
- Данные кодируются только в формате JSON, что добавляет все недостатки JSON.
- Возможно усложнение пользовательской аналитики, поскольку запросы пользователей API немного различаются. В REST и RPC сразу видно, сколько запросов делается к каждой конечной точке API, но это менее очевидно в GraphQL.
- Использовать GraphQL для внешних API необходимо с осторожностью. Это все равно что предоставить доступ к базе данных и разрешить клиентам отправлять запросы SQL.

Многие преимущества GraphQL могут быть реализованы в REST. Простой API может начинаться с простых REST-методов HTTP (GET, POST, PUT, DELETE) с простыми телами JSON. По мере усложнения требований появляется возможность использования более широких возможностей REST (таких, как OData https://www.odata.org/) или функциональности JSON-API (например, https://jsonapi.org/format/#fetching-includes) для объединения взаимосвязанных данных из нескольких источников в один запрос. GraphQL может быть удобнее REST для сложных требований, поскольку GraphQL предоставляет стандартную реализацию и документирование его функциональности. У REST, в отличие от него, единого стандарта нет.

6.7.5. WebSocket

WebSocket — коммуникационный протокол для полнодуплексной передачи данных по долгосрочному подключению TCP в отличие от протокола HTTP, который создает новое подключение для каждого запроса и закрывает его с каждым ответом. REST, RPC, GraphQL и модель Actor — паттерны проектирования или философии, тогда как WebSocket и HTTP являются коммуникационными протоколами. Тем не менее разумно сравнивать WebSocket с REST как архитектурные стили, поскольку API можно реализовать с использованием WebSocket вместо других четырех вариантов.

Чтобы создать подключение WebSocket, клиент отправляет запрос WebSocket к серверу. WebSocket использует handshake HTTP для создания исходного подключения и обращается с запросом к сервису для перехода на WebSocket с HTTP. Последующие сообщения могут использовать WebSocket на базе долгосрочного подключения TCP.

WebSocket поддерживает подключения открытыми, что повышает накладные расходы для всех сторон. Это означает, что WebSocket имеет состояние (в отличие от REST и HTTP, не имеющих состояния). Запрос должен обрабатываться хостом, который содержит актуальное состояние/подключение, в отличие от REST, где любой хост может обрабатывать любой запрос. WebSocket — это протокол с сохранением состояния; кроме того, существуют затраты на поддержание подключений. Эти два фактора означают, что WebSocket хуже масштабируется.

WebSocket допускает взаимодействия p2p, так что бэкенд не требуется. Он жертвует масштабируемостью ради снижения задержки и повышения производительности.

6.7.6. Сравнение

В ходе собеседования вам могут предложить сравнить плюсы и минусы этих архитектурных стилей и факторы, которые необходимо учитывать при выборе стиля и протокола. REST и RPC — самые распространенные варианты. Стартапы

для простоты обычно используют REST, тогда как большие организации могут выиграть от эффективности и прямой/обратной совместимости RPC. GraphQL — относительно новая философия. Протокол WebSocket полезен для двусторонних коммуникаций, включая коммуникации p2p. За дополнительной информацией обращайтесь к https://apisyouwonthate.com/blog/picking-api-paradigm/ и https://www.baeldung.com/rest-vs-websockets.

ИТОГИ

- Шлюз API — веб-сервис, который проектировался как облегченное решение без сохранения состояния, при этом обеспечивающее многие виды сквозной функциональности для многих сервисов. Функции API можно разделить на относящиеся к категориям безопасности, проверки ошибок, производительности и доступности, а также ведения журналов.
- Сервисная сеть, или паттерн sidecar, — альтернативный паттерн. Каждый хост получает собственный sidecar-контейнер, так что никакой сервис не сможет потребить несправедливую долю ресурсов.
- Для минимизации сетевого трафика можно воспользоваться сервисом метаданных для хранения информации, обрабатываемой многими компонентами в системе.
- Обнаружение сервисов — механизм, позволяющий клиентам узнать, какие хосты сервисов доступны.
- Браузерное приложение может иметь два или больше сервиса бэкенда. Один из них — веб-сервер, который перехватывает запросы и ответы от других внутренних сервисов.
- Сервис веб-сервера минимизирует сетевой трафик между браузером и датацентром, выполняя операции агрегирования и фильтрации на бэкенде.
- Фреймворки браузерных приложений предназначены для разработки браузерных приложений. Серверные фреймворки предназначены для разработки веб-сервисов. Разработка мобильных приложений может выполняться с использованием нативных или кросс-платформенных фреймворков.
- Существуют кросс-платформенные и полностековые фреймворки для разработки браузерных приложений, мобильных приложений и веб-серверов. У этих фреймворков есть свои недостатки, из-за которых они могут оказаться неподходящими для конкретных требований.
- Бэкенд-фреймворки можно разделить на фреймворки, использующие технологии RPC, REST и GraphQL.
- Некоторые компоненты могут быть реализованы как в виде библиотек, так и в виде сервисов. У каждого подхода есть свои плюсы и минусы.

- Многие коммуникационные парадигмы реализуются на базе HTTP. RPC — низкоуровневый протокол для обеспечения эффективности.
- Архитектура REST проста в изучении и использовании. Ресурсы REST следует объявлять как кэшируемые там, где это возможно.
- REST требует отдельного фреймворка документирования, такого как OpenAPI.
- RPC — двоичный протокол, спроектированный для оптимизации ресурсов. Его правила изменения схем также допускают прямую и обратную совместимость.
- GraphQL поддерживает точечные запросы и содержит интегрированные средства документирования API. С другой стороны, он сложен и его защита требует больших усилий.
- WebSocket — коммуникационный протокол с сохранением состояния для полнодуплексной передачи данных. Он требует более высоких накладных расходов на стороне клиента и сервера по сравнению с другими коммуникационными парадигмами.

Часть 2

В части 1 были рассмотрены темы, часто встречающиеся на собеседованиях по проектированию систем. В этой главе мы перейдем к примерным задачам, которые могут быть вам поставлены. В каждом решении применяются концепции, которые рассматривались в части 1, а также концепции, относящиеся к конкретному вопросу.

Начнем с главы 7, в которой будет рассматриваться система, похожая на Craigslist, оптимизированная для простоты обсуждения.

В главах 8–10 рассматривается дизайн систем, которые становятся частью многих других систем.

В главе 11 обсуждается сервис автозавершения / опережающего ввода — типичная система, которая непрерывно поглощает и обрабатывает большие объемы данных в структуру данных объемом несколько мегабайт и к которой пользователи обращаются с целевыми запросами.

В главе 12 говорится о сервисе общего доступа к изображениям. Общий доступ и взаимодействие с графикой и видео являются основной функциональностью практически всех социальных сетей, и эта тема часто затрагивается на собеседованиях. Это приводит нас к теме главы 13, где рассматривается CDN (Content Distribution Network) — система, обычно используемая для экономически эффективного предоставления статического контента (такого, как графика и видео) глобальной аудитории.

В главе 14 рассматривается приложение для передачи текстовых сообщений — система доставки сообщений для большого количества пользователей с предотвращением случайной доставки дубликатов.

В главе 15 обсуждается система бронирования жилья, в которой арендодатели предлагают жилье для аренды, а наниматели бронируют и оплачивают его. Система также должна предоставлять доступ внутренним пользователям для проведения арбитража и модерирования контента.

В главах 16 и 17 рассматриваются системы лент данных. В главе 16 обсуждается система ленты новостей, которая сортирует данные, доставляя пользователям интересующий их контент, тогда как в главе 17 — сервис аналитики данных, агрегирующий большие объемы данных на дашборде, который может использоваться для принятия решений.

7 Дизайн Craigslist

В ЭТОЙ ГЛАВЕ

- ✓ Проектирование приложения с двумя разными типами пользователей
- ✓ Геолокационная маршрутизация для группировки пользователей
- ✓ Проектирование приложений с интенсивным чтением и интенсивной записью
- ✓ Другие возможные темы для обсуждения на собеседовании

Вам поручено спроектировать веб-приложение для классифицированных постов (сообщений). Craigslist — пример типичного веб-приложения, которое может иметь более миллиарда пользователей. Пользователи группируются по географическому признаку. Можно обсудить систему в целом: браузер и мобильные приложения, бэкенд без сохранения состояния, простые требования к хранению данных и аналитику. В задачу могут быть добавлены другие сценарии использования и ограничения для открытого обсуждения. Это единственная глава, в которой мы обсудим монолитную архитектуру как возможный дизайн системы.

7.1. ПОЛЬЗОВАТЕЛЬСКИЕ ИСТОРИИ И ТРЕБОВАНИЯ

Рассмотрим пользовательские истории для Craigslist. Мы будем различать две основные категории пользователей: читатели и авторы.

Автор должен иметь возможность создавать и удалять посты, а также искать свои посты, которых может быть достаточно много (особенно если они были сгенерированы на программном уровне). Пост должен содержать следующую информацию:

- заголовок;
- несколько абзацев с описанием;
- цена. Предполагается одна валюта без возможности конвертации;
- местонахождение;
- до 10 фотографий не более 1 Мбайт каждая;
- видео, хотя оно может быть добавлено на последующей итерации приложения.

Автор может продлевать действие поста каждую неделю. Он получает извещения по электронной почте со ссылкой для продления поста.

Читателю должны быть доступны следующие функции:

1. Просмотр всех постов или поиск в постах из любого города за последние 7 дней. Просмотр списка результатов (возможно, с бесконечной прокруткой).
2. Применение фильтров к результатам.
3. Выбор отдельного поста для подробного просмотра.
4. Связь с автором (возможно, по электронной почте).
5. Оповещение о попытках мошенничества и постах, содержащих неверную информацию (один из приемов кликбейта — указание низкой цены в заголовке поста с более высокой ценой в описании).

Нефункциональные требования:

- *Масштабируемость* — до 10 миллионов пользователей из одного города.
- *Высокая доступность* — работоспособность 99,9%.
- *Высокая производительность* — читатели должны иметь возможность просматривать пост через несколько секунд после его создания. Значение 99-го процентиля для поиска и просмотра постов должно быть равно 1 секунде.
- *Безопасность* — автор должен войти в систему перед созданием постов. Для этого можно использовать библиотеку или сервис аутентификации. В при-

ложении Б обсуждается OpenID Connect — популярный механизм аутентификации. В этой главе данная тема не рассматривается.

Бóльшая часть пространства хранения данных будет использоваться для постов Craigslist. Объем необходимого хранилища будет низким:

- Возможно, вы решите показывать пользователю Craigslist только посты, относящиеся к его местоположению. Это означает, что в датацентре, обслуживающем любого конкретного пользователя, должна храниться лишь малая часть всех постов (хотя в нем также могут храниться резервные копии постов из других датацентров).
- Посты создаются вручную (а не генерируются на программном уровне), так что увеличение затрат памяти будет медленным.
- Система не должна обрабатывать данные, сгенерированные на программном уровне.
- Пост может быть автоматически удален через неделю.

Низкие требования к хранилищу данных означают, что все данные могут поместиться на одном хосте, поэтому решение распределенного хранения данных не потребуется. Допустим, средний пост содержит 1000 букв, или около 1 Кбайт текста. Если предположить, что в большом городе проживает 10 миллионов человек и 10 % из них — авторы, создающие в среднем 10 постов в день (то есть 10 Гбайт в день), в базе данных SQL легко могут храниться посты за несколько месяцев.

7.2. API

Набросаем черновую схему конечных точек API, разделенных на категории управления постами и управления пользователями. (На собеседовании у вас не будет времени на составление формальной спецификации API, например, в формате OpenAPI или GraphQL; вы можете сказать эксперту, что для определения API можно воспользоваться формальной спецификацией, но для экономии времени будете использовать примерные наброски. К этому вопросу мы больше возвращаться не будем.)

Посты CRUD:

- `GET` и `DELETE /post/{id}`
- `GET /post?search={search_string}`. Может быть конечной точкой для получения всех постов запросом `GET`. Также может поддерживать параметр запроса `search` для поиска по содержимому постов. Кроме того, можно реализовать параметры запроса для получения данных по страницам (эта возможность рассматривается в разделе 12.7.1).
- `POST` и `PUT /post`

- `POST /contact`
- `POST /report`
- `DELETE /old_posts`

Управление пользователями:

- `POST /signup`. Мы не будем обсуждать управление учетными записями пользователей.
- `POST /login`
- `DELETE /user`

Прочие:

- `GET /health`. Обычно генерируется фреймворком автоматически. Реализация может быть как очень простой, с выдачей короткого запроса GET и проверкой того, что для него возвращается код 200, так и подробной, с включением такой статистики, как 99-й процентиль, и информации о доступности разных конечных точек.

Приложение содержит фильтры, которые могут изменяться в зависимости от категории товара. Для простоты предполагается использование фиксированного набора фильтров. Фильтры могут быть реализованы как на фронтенд, так и на бэкенд:

- *Местоположение*: `enum`.
- *Минимальная цена*.
- *Максимальная цена*.
- *Состояние товара*: `enum` со значениями `NEW` (Новое), `EXCELLENT` (Отличное), `GOOD` (Хорошее) и `ACCEPTABLE` (Удовлетворительное).

Конечная точка `GET /post` может поддерживать параметр запроса `search` для поиска по содержимому постов.

7.3. СХЕМА БАЗЫ ДАННЫХ SQL

Для хранения данных пользователей и постов Craigslist можно спроектировать следующую схему SQL:

- *Пользователь*: `id PRIMARY KEY`, `first_name text`, `last_name text`, `signup_ts integer`.
- *Пост*: таблица денормализована, поэтому для получения всей информации поста не придется выполнять запросы `JOIN`. `id PRIMARY KEY`, `created_at integer`, `poster_id integer`, `location_id integer`, `title text`, `description text`, `price integer`, `condition text`, `country_code char(2)`, `state text`, `city`

`text`, `street_number integer`, `street_name text`, `zip_code text`, `phone_number integer`, `email text`

- *Графика*: `id PRIMARY KEY`, `ts integer`, `post_id integer`, `image_address text`
- *Жалобы*: `id PRIMARY KEY`, `ts integer`, `post_id integer`, `user_id integer`, `abuse_type text`, `message text`
- *Хранение изображений*: изображения могут храниться в хранилище объектов. Среди них особенно популярны AWS S3 и Azure Blob Storage, поскольку они надежны, просты в использовании и обслуживании и экономичны.
- `image_address`: идентификатор, используемый для загрузки изображения из хранилища объектов.

Если требуется низкая задержка (например, при реакции на запросы пользователей), обычно используются базы данных SQL или базы данных в памяти с низкой задержкой, такие как Redis. Базы данных NoSQL, использующие распределенные файловые системы (такие, как HDFS), предназначены для больших заданий обработки данных.

7.4. ИСХОДНАЯ ВЫСОКОУРОВНЕВАЯ АРХИТЕКТУРА

Для первого дизайна Craigslist, представленного на рис. 7.1, можно обсудить два варианта в порядке возрастания сложности. Они более подробно рассматриваются в следующих двух разделах.

1. Монолит, использующий сервис аутентификации пользователей для выполнения аутентификации и хранилище объектов для хранения постов.
2. Клиентский сервис, бэкенд, сервис SQL, хранилище объектов и сервис аутентификации пользователей.

В оба варианта также включается сервис ведения журналов, поскольку он необходим для эффективной отладки системы. Для простоты исключим из архитектуры мониторинг и оповещения. Тем не менее многие облачные провайдеры предоставляют средства ведения журналов, мониторинга и оповещения, которые легко настраиваются; их стоит использовать.

7.5. МОНОЛИТНАЯ АРХИТЕКТУРА

Вариант с использованием монолита не очевиден, он даже может привести эксперта в замешательство. Вам вряд ли когда-нибудь придется использовать монолитную архитектуру веб-сервисов в своей работе. Однако не стоит забывать, что все проектировочные решения сводятся к их достоинствам и недостаткам, поэтому вы не должны бояться предлагать такие решения и обсуждать связанные с ними компромиссы.

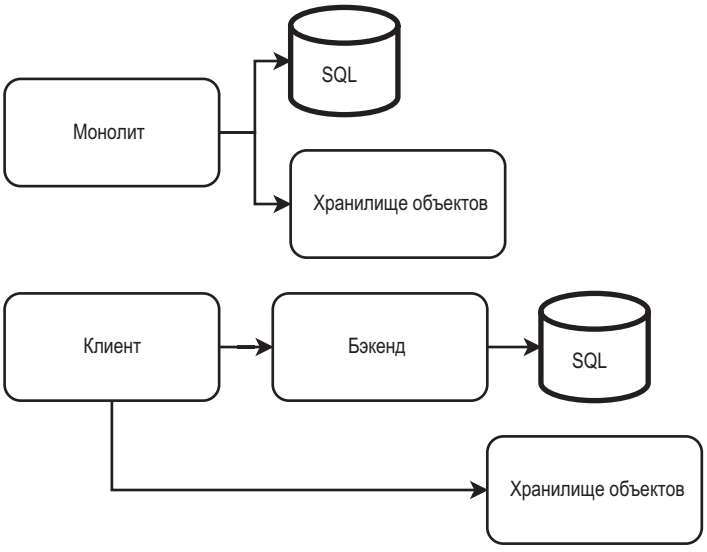

Рис. 7.1. Простые варианты исходной архитектуры. Сверху: высокоуровневая архитектура состоит из монолита и хранилища объектов. Снизу: традиционная высокоуровневая архитектура с UI-фронтенд-сервисом и бэкенд-сервисом. Файлы изображений хранятся в хранилище объектов, к которому клиенты обращаются с запросами. Остальные данные постов хранятся в SQL

Наше приложение можно реализовать в виде монолита, содержащего UI и функциональность бэкенда, и хранить целые веб-страницы в хранилище объектов. Ключевое проектировочное решение заключается в том, что вся веб-страница поста может полностью храниться в хранилище объекта, включая фотографии. Это означает, что многие столбцы таблицы постов, описанные в разделе 7.3, не понадобятся; таблица будет использоваться для высокоуровневой архитектуры, представленной в нижней части рис. 7.1, мы рассмотрим ее далее в этой главе.

На рис. 7.2 домашняя страница статична, не считая навигационной панели (которая содержит информацию о местонахождении вида SF bay area, ссылки на более конкретные места: sfc, sby и т. д.) и раздел nearby cl со списком других городов. Другие части статичны, включая ссылки на левой навигационной панели (например, craigslist app и about craigslist) и ссылки на нижней навигационной панели: help, safety, privacy и т. д.

Такое решение просто реализуется и сопровождается. Его основные недостатки:

1. Теги HTML, CSS и JavaScript дублируются в каждом посте.
2. Если вы разрабатываете нативное мобильное приложение, оно не сможет использовать один и тот же бэкенд с браузерным приложением. Одно из возможных решений — прогрессивное веб-приложение (см. раздел 6.5.3),

которое устанавливается на мобильных устройствах и может использоваться в веб-браузерах на любых устройствах.

3. Любая аналитика постов потребует парсинга HTML. Впрочем, это не самый серьезный недостаток. Вы можете создавать и поддерживать собственные вспомогательные скрипты для выборки страниц постов и парсинга HTML.

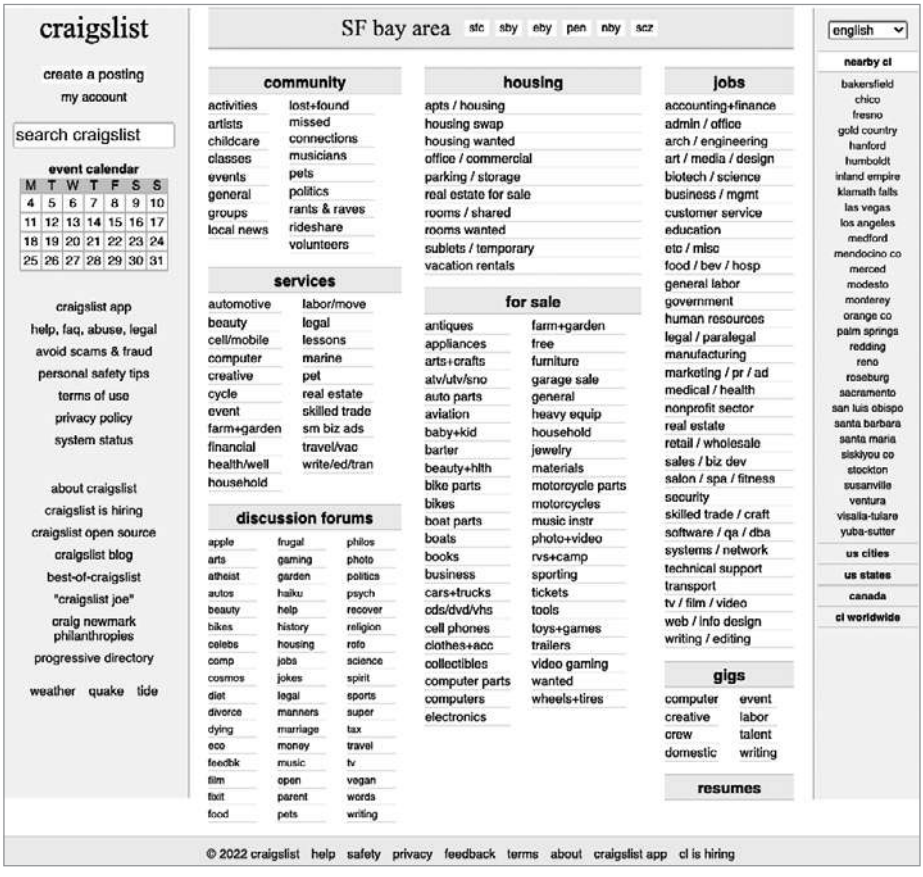

Рис. 7.2. Домашняя страница Craigslist (источник: https://sfbay.craigslist.org/)

Недостаток первого решения заключается в том, что хранение дублирующихся компонентов страниц потребует дополнительной памяти. Другим недостатком станет невозможность применения новой функциональности или полей к старым постам, хотя, поскольку посты автоматически удаляются через одну неделю, это может быть допустимо в зависимости от требований. Вы можете обсудить этот вариант с экспертом как пример того, что при проектировании системы не стоит предполагать неявных требований.

Второй недостаток можно частично компенсировать написанием браузерного приложения с использованием отзывчивого дизайна и реализацией мобильного приложения как обертки для браузерного приложения с использованием компонентов WebView. https://github.com/react-native-webview/react-native-webview — библиотека WebView для React Native. https://developer.android.com/reference/android/webkit/WebView — библиотека WebView для Android, а https://developer.apple.com/documentation/webkit/wkwebview — библиотека WebView для iOS. Можно использовать медиазапросы CSS (https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Responsive_Design#media_queries) для вывода различных макетов страниц на экраны телефонов, планшетов, портативных и настольных компьютеров. В таком случае вам не придется использовать UI-компоненты из мобильного фреймворка. Сравнение интерфейсов, созданных с применением этого подхода и традиционного подхода с использованием UI-компонентов во фреймворках мобильной разработки, выходит за рамки книги.

Для выполнения аутентификации в бэкенд-сервисе и сервисе хранения объектов можно воспользоваться сторонним сервисом аутентификации пользователей или разработать собственный сервис. Механизмы аутентификации Simple Login и OpenID Connect подробно рассматриваются в приложении Б.

7.6. РАБОТА С БАЗОЙ ДАННЫХ SQL И ХРАНИЛИЩЕМ ОБЪЕКТОВ

На нижней диаграмме рис. 7.1 изображена более традиционная высокоуровневая архитектура. UI-фронтенд-сервис обращается с запросами к бэкенд-сервису и сервису хранения объектов. Бэкенд-сервис обращается с запросами к сервису SQL.

При таком подходе хранилище объектов используется для файлов изображений, а в базе данных SQL хранятся остальные данные постов, как говорилось в разделе 7.4. Все данные, включая изображения, можно было бы хранить в базе данных SQL и вообще обойтись без хранилища объектов. Однако это будет означать, что клиенту придется загружать файлы изображений через хост бэкенда. Это создает дополнительную нагрузку на хост бэкенда и увеличивает задержку при загрузке изображений и общую вероятность ошибок загрузки из-за внезапных проблем с сетевыми соединениями.

Если вы хотите сохранить простоту исходной реализации, можно изначально отказаться от добавления изображений в посты и добавить хранилище объектов, когда вы будете реализовывать эту возможность.

Как бы то ни было, поскольку каждый пост ограничен 10 файлами изображений по 1 Мбайт каждый, а хранить большие файлы нежелательно, можно обсудить с экспертом вероятность того, что это требование изменится в будущем. Если изображения большего размера, скорее всего, не потребуются, то файлы

изображений можно хранить в SQL. В таблицу изображений можно добавить текстовый столбец `post_id` и `BLOB`-столбец с данными изображения. Преимуществом такого решения является его простота.

7.7. ТРУДНОСТИ С МИГРАЦИЕЙ

Раз уж речь зашла о выборе подходящих хранилищ данных для нефункциональных требований, обсудим проблему миграции данных, прежде чем переходить к другим возможностям и требованиям.

У хранения файлов изображений в базах данных SQL есть еще один недостаток: в будущем может потребоваться провести миграцию и сохранить их в хранилище объектов. Как правило, миграция с переходом на другое хранилище данных — дело утомительное и нудное.

Обсудим возможный простой процесс миграции, приняв следующие предположения:

1. Оба хранилища данных могут интерпретироваться как отдельные сущности. Это означает, что процесс репликации абстрагирован, и для оптимизации нефункциональных требований (таких, как задержка или доступность) не нужно думать о том, как данные распределены между датацентрами.
2. Временная неработоспособность допустима. Можно запретить запись в приложении на время миграции данных, чтобы пользователи не добавляли новые данные в старое хранилище в процессе переноса данных из старого хранилища в новое.
3. В начале периода неработоспособности можно отключать/завершать запросы, чтобы пользователи, отправляющие запросы на запись (POST, PUT, DELETE), получали ошибку 500. Можно заранее оповестить пользователей о предстоящих ограничениях по различным каналам: электронной почте, в браузере, push-уведомлениях на мобильных устройствах или с использованием баннеров в клиенте.

Можно написать скрипт Python, который запускается на ноутбуке разработчика, читает записи из одного хранилища и записывает их в другое. Как показано на рис. 7.3, этот скрипт отправляет запросы GET на бэкенд для получения текущих записей данных и запросы POST для сохранения в новом хранилище данных. В общем случае этот простой метод подходит, если передача данных может завершиться за несколько часов и ее достаточно выполнить однократно. На написание такого скрипта потребуется несколько часов, и возможно, на его доработку в целях ускорения передачи данных больше времени тратить не стоит.

Следует ожидать, что задание миграции может внезапно остановиться из-за ошибок или сетевых проблем и выполнение скрипта придется перезапустить. Конечные точки записи должны быть идемпотентными, чтобы предотвратить запись

дубликатов в новое хранилище данных. Скрипт должен создавать контрольные точки, чтобы уже переданные записи не приходилось читать и записывать заново. Простого механизма контрольных точек будет достаточно; после каждой записи идентификатор объекта сохраняется локально на жестком диске. Если при выполнении задания произойдет сбой, задание можно продолжить с контрольной точки при перезапуске (после исправления ошибок, если понадобится).

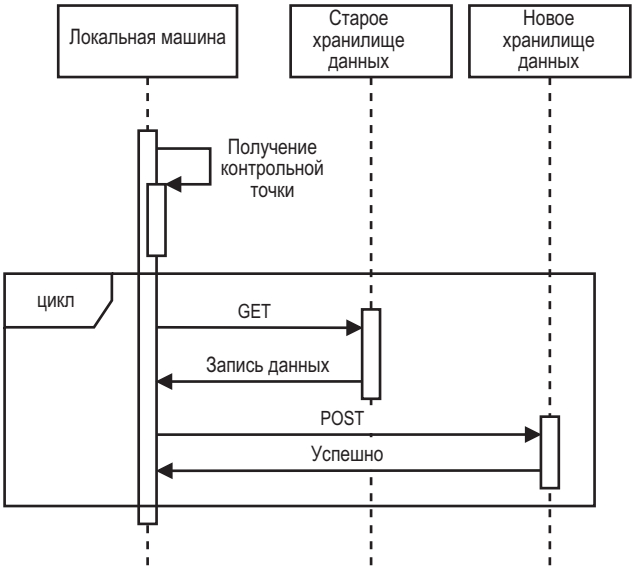

Рис. 7.3. Диаграмма последовательности действий простого процесса миграции данных. Локальная машина сначала находит контрольную точку, если она есть, а затем отправляет необходимые запросы для перемещения каждой записи из старого хранилища данных в новое

Внимательный читатель заметит, что для работы механизма контрольных точек скрипт должен читать записи в одинаковом порядке при каждом запуске. Это можно сделать разными способами, включая следующие:

- Если вы можете получить полный и отсортированный список идентификаторов записей и сохранить его на жестком диске локальной машины, скрипт может загрузить этот список в память перед началом передачи данных. Скрипт получает каждую запись по идентификатору, записывает ее в новое хранилище данных и сохраняет на жесткий диск информацию о том, что этот идентификатор передан. Так как запись на жесткий диск производится медленно, можно записывать/создавать контрольные точки для этих завершенных идентификаторов группами (пакетами). В пакетном режиме в задании может произойти сбой, прежде чем пакет идентификаторов пройдет контрольную точку, и эти объекты придется заново прочитать и записать; идемпотентные контрольные точки записи предотвратят дублирование записей.
- Если в объектах данных присутствуют порядковые (определяющие позицию) поля (например, метки времени), скрипт может использовать эти поля при

создании контрольных точек. Например, если контрольные точки упорядочиваются по дате, скрипт сначала передает записи с самой ранней датой, создает контрольную точку для этой даты, увеличивает дату, передает записи с этой датой и т. д., пока вся передача не будет завершена.

Скрипт должен прочитать/записать поля объектов данных в соответствующие таблицы и столбцы. Чем больше функциональности будет добавлено перед миграцией, тем сложнее будет миграционный скрипт. Новая функциональность означает новые классы и свойства. В базе данных появляются новые таблицы и столбцы, вам придется создавать большое количество запросов ORM/SQL, команды запросов усложнятся, и могут потребоваться соединения таблиц инструкцией `JOIN`.

Если передача данных слишком масштабна, чтобы ее можно было завершить этим методом, придется запускать скрипт в датацентре. Его можно запустить отдельно на каждом хосте, если данные распределены между хостами. Использование нескольких хостов позволяет выполнить миграцию данных без простоев. Если хранилище данных распределено между многими хостами, значит, у системы много пользователей, и в этом случае ее неработоспособность будет стоить прибыли и репутации.

Чтобы вывести из эксплуатации старое хранилище данных постепенно, по одному хосту, на каждом хосте выполняются следующие действия.

1. На хосте исчерпываются все подключения. Иначе говоря, существующие запросы завершаются, а новые не принимаются. Узнать больше об исчерпании подключений можно здесь: https://cloud.google.com/load-balancing/docs/enabling-connection-draining, https://aws.amazon.com/blogs/aws/elb-connection-draining-remove-instances-from-service-with-care/ и здесь: https://docs.aws.amazon.com/elasticloadbalancing/latest/classic/conig-conn-drain.html.
2. После того как запросы будут исчерпаны, на хосте запускается скрипт передачи данных.
3. После того как скрипт завершит работу, этот хост не понадобится.

Как поступать в случае ошибок записи? Если миграция занимает несколько часов или дней, нельзя допускать, чтобы задание передачи данных аварийно завершалось при любой ошибке чтения или записи данных. Вместо этого скрипт должен сохранять информацию об ошибках и продолжать выполнение. Каждый раз, когда происходит ошибка, вы регистрируете запись, с которой она возникла, в журнале и продолжаете чтение и запись других данных. Затем информацию можно просмотреть, исправить ошибки в случае необходимости и заново выполнить скрипт для передачи этих конкретных записей.

Из всего сказанного следует вывод: миграция данных сложна и затратна, и ее желательно по возможности избегать. Подходящие для системы хранилища данных

лучше выбирать с самого начала, при ее проектировании, а не мигрировать на них позднее, если только вы не реализуете систему в экспериментальных целях для обработки небольшого объема данных (желательно некритичных, которые можно потерять без последствий).

7.8. ЗАПИСЬ И ЧТЕНИЕ ПОСТОВ

На рис. 7.4 изображена последовательная диаграмма создания поста автором в архитектуре, описанной в разделе 7.6. Хотя мы записываем данные сразу в несколько сервисов, средства обеспечения согласованности в распределенных транзакциях не требуются. Происходит следующее:

1. Клиент направляет запрос POST на бэкенд без добавления изображений. Бэкенд записывает пост в базу данных SQL и возвращает идентификатор поста клиенту.
2. Клиент может отправить файлы изображений в хранилище объектов последовательно или инициировать потоки для создания запросов на параллельную отправку.

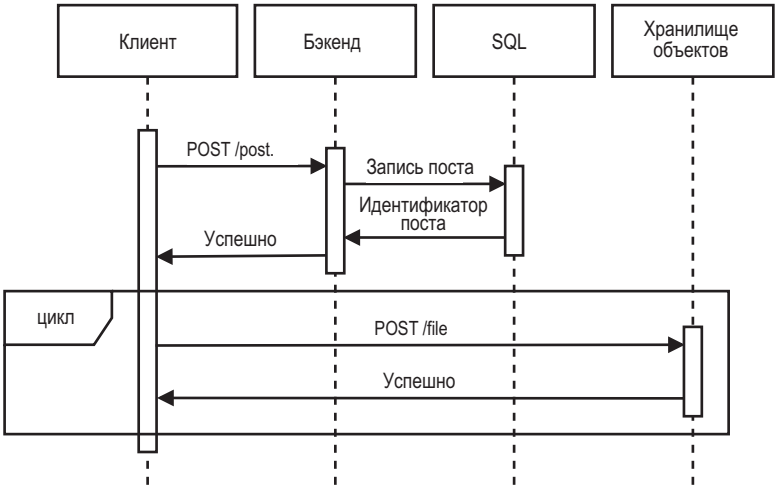

Рис. 7.4. Диаграмма последовательности действий при создании нового поста, при котором за отправку изображений отвечает клиент

При таком подходе бэкенд возвращает код успеха 200, не зная, удалось ли успешно отправить файлы изображений в хранилище объектов. Чтобы бэкенд знал, что весь пост отправлен успешно, он должен отправить изображения в хранилище объектов самостоятельно.

На рис. 7.5 показан пример такого подхода. Бэкенд возвращает клиенту код успеха 200 только после того, как все файлы изображений будут успешно отправлены в хранилище объектов — на случай, если отправка завершится неудачей. Это может произойти по разным причинам, например из-за сбоя хоста бэкенда в процессе отправки, проблем с сетевыми подключениями или недоступности хранилища объектов.

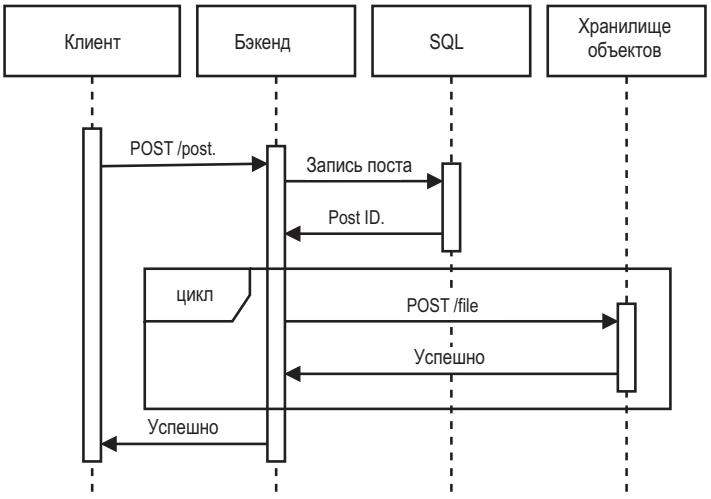

Рис. 7.5. Диаграмма последовательности действий при создании нового поста, при котором за отправку изображений отвечает бэкенд

Рассмотрим плюсы и минусы обоих подходов. Преимущества исключения бэкенда из отправки изображений:

- *Сниженное потребление ресурсов* — бремя отправки изображений возлагается на клиент. Если отправка выполняется через бэкенд, то бэкенд должен масштабироваться вместе с хранилищем объектов.
- *Снижение общей задержки* — изображениям не нужно проходить через дополнительный хост. Если вы решите использовать CDN для хранения изображений, проблема с задержкой дополнительно усугубится, поскольку клиенты не смогут пользоваться ресурсами CDN, находящимися поблизости.

Преимущества включения бэкенда в отправку изображений:

- Не нужно реализовывать и поддерживать механизмы аутентификации и авторизации в хранилище объектов. Так как хранилище объектов недоступно для окружающего мира, сокращается общая поверхность атаки системы.
- Читатели гарантированно смогут просмотреть все изображения из поста. В предыдущем решении, если некоторые изображения не удалось успешно отправить, читатели не увидят их при просмотре постов. Вы можете обсудить с экспертом, допустим ли такой компромисс.

Одно из решений, сочетающее достоинства описанных выше вариантов, — запись изображений на бэкенд и чтение их из CDN.

ВОПРОС Какими достоинствами и недостатками обладает отправка каждого изображения в отдельном запросе по сравнению с отправкой всех файлов в одном запросе?

Действительно ли клиенту необходимо отправлять каждое изображение в отдельном запросе? Такая сложность может оказаться излишней. Максимальный размер запроса на запись чуть выше 10 Мбайт; это достаточно мало, и данные передаются за считаные секунды. Но это также означает, что повторные попытки будут более затратными. Обсудите эти нюансы с экспертом.

Диаграмма последовательности действий читателя при чтении поста идентична рис. 7.4, не считая того, что вместо запросов POST используются GET. Когда читатель читает пост, бэкенд извлекает пост из базы данных SQL и возвращает его клиенту. После этого клиент получает и отображает графические изображения поста из хранилища объектов. Запросы на выборку изображений могут быть параллельными, так что файлы хранятся на разных хостах и могут реплицироваться, что позволяет загружать их параллельно с разных хостов.

7.9. ФУНКЦИОНАЛЬНОЕ СЕКЦИОНИРОВАНИЕ

Первый шаг при масштабировании — применение функционального секционирования по географическому признаку, например по городу. Обычно этот процесс называется *геолокационной маршрутизацией* — обслуживанием трафика на основании региона, из которого исходят запросы DNS в зависимости от географического местоположения пользователей. Приложение можно развернуть в нескольких датацентрах и маршрутизировать каждого пользователя к датацентру, обслуживающему его город; обычно это будет ближайший датацентр. Таким образом, кластер SQL в каждом датацентре хранит только данные обслуживаемых им городов. Можно реализовать репликацию каждого кластера SQL на двух других сервисах SQL в разных датацентрах, в соответствии с описанием репликации на базе binlog для MySQL (см. раздел 4.3.2).

Craigslist реализует географическое секционирование, назначая каждому городу субдомен (например, `sfbay.craigslist.org`, `shanghai.craiglist.org` и т. д.). При открытии `craigslist.org` в браузере выполняются следующие действия (пример показан на рис. 7.6):

1. Интернет-провайдер проводит сопоставление DNS для адреса `craigslist.org` и возвращает его IP-адрес (в браузерах и ОС присутствуют кэши DNS, так что браузер может использовать свой кэш DNS или кэш DNS ОС для

будущих поисков DNS, что быстрее, чем отправка запроса на сопоставление DNS интернет-провайдеру).

2. Браузер отправляет запрос к IP-адресу `craigslist.org`. Сервис определяет местоположение на основании IP-адреса и возвращает ответ 3xx с субдоменом, соответствующим местонахождению браузера. Возвращенный адрес может кэшироваться браузером и другими посредниками (например, ОС пользователя и интернет-провайдером).
3. Для получения IP-адреса этого субдомена потребуется еще один поиск DNS.
4. Браузер выдает запрос к IP-адресу субдомена. Сервис возвращает веб-страницу и данные этого субдомена.

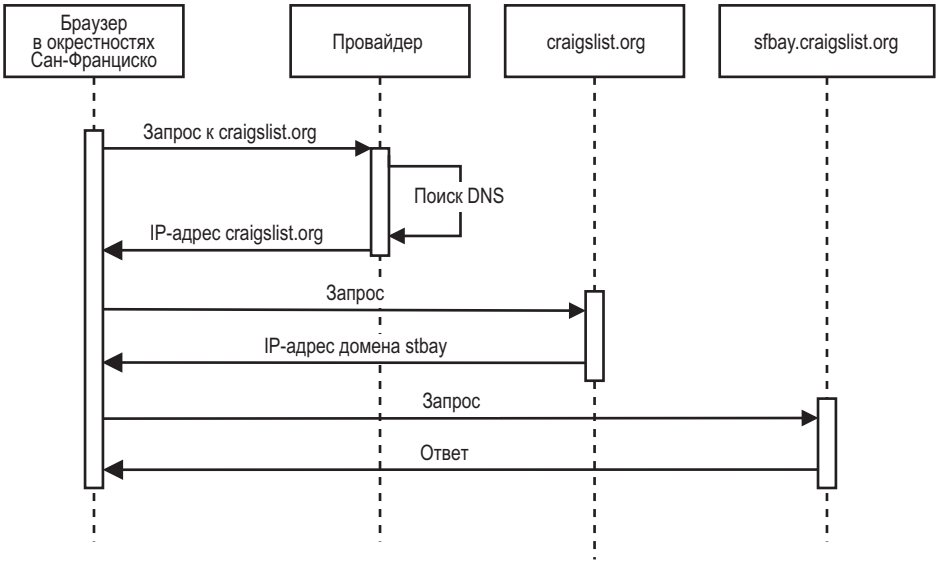

Рис. 7.6. Диаграмма последовательности действий при использовании GeoDNS для перенаправления пользовательских запросов по соответствующим IP-адресам

Для Craigslist можно воспользоваться GeoDNS. Браузеру достаточно выполнить поиск DNS однократно для `craigslist.org`, а возвращаемый IP-адрес будет принадлежать датацентру, соответствующему местонахождению браузера. После этого браузер может выдать запрос к этому центру данных, чтобы получить посты города. Вместо того чтобы указывать субдомен в адресной строке браузера, можно выбрать город в раскрывающемся меню пользовательского интерфейса приложения. Пользователь выбирает город, чтобы отправить запрос соответствующему датацентру и просмотреть посты этого города. В интерфейсе также может присутствовать простая статическая веб-страница со всеми городами Craigslist; пользователь может выбрать нужный город щелчком на ссылке с названием.

Облачные сервисы, такие как AWS (https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/routing-policy-geo.html), предоставляют руководства по настройке геолокационной маршрутизации.

7.10. КЭШИРОВАНИЕ

Некоторые посты могут стать очень популярными, и на них приходится большая доля запросов на чтение; например, пост с предложением товара по цене значительно ниже рыночной. Для соблюдения SLA по задержке (например, 1-секундное значение 99-го процентиля) и предотвращения ошибок тайм-аута 504 популярные посты можно кэшировать.

Кэш LRU можно реализовать на базе Redis. Ключом может быть идентификатор поста, а значением — вся страница HTML поста. Сервис изображений можно реализовать перед хранилищем объектов, чтобы он поддерживал собственный кэш, связывающий идентификаторы объектов с изображениями.

Статическая природа постов ограничивает потенциальное устаревание кэша, хотя автор может обновить свой пост. В таком случае хост должен обновить соответствующий элемент кэша.

7.11. CDN

На рис. 7.7 изображена схема с использованием CDN, хотя Craigslist содержит очень мало статического контента (изображений и видео), который виден поль-

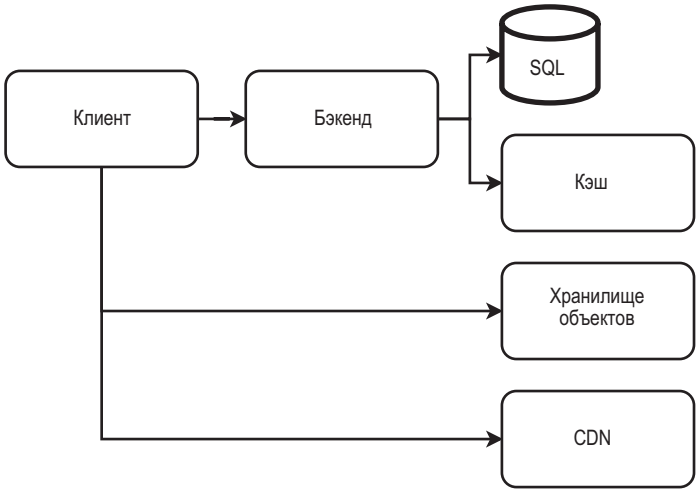

Рис. 7.7. Диаграмма архитектуры Craigslist после добавления кэша и CDN

зователям. В основном статический контент образуют файлы CSS и JavaScript, которые занимают лишь несколько мегабайт. Для файлов CSS и JavaScript также можно применить браузерное кэширование (механизм браузерного кэширования рассмотрен в разделе 4.10).

7.12. МАСШТАБИРОВАНИЕ ЧТЕНИЯ С ИСПОЛЬЗОВАНИЕМ КЛАСТЕРА SQL

Маловероятно, что обсуждение выйдет за рамки функционального секционирования и кэширования. При необходимости масштабировать чтение используйте методы, описанные в главе 3, одним из которых является репликация SQL.

7.13. МАСШТАБИРОВАНИЕ ПРОПУСКНОЙ СПОСОБНОСТИ ЗАПИСИ

В начале главы упоминалось о том, что создаваемое приложение является интенсивным по чтению. Вам вряд ли когда-нибудь потребуется генерировать посты на программном уровне. В этом разделе описана гипотетическая ситуация, при которой система дает пользователям такую возможность и предоставляет общедоступный API для создания постов.

Если при вставке и обновлении на хосте SQL будут происходить выбросы трафика, необходимая пропускная способность может превысить максимальную. Согласно https://stack-overlow.com/questions/2861944/how-to-do-very-fast-inserts-to-sql-server-2008, некоторые реализации SQL предоставляют методы для быстрого выполнения `INSERT`: например, скорость `ExecuteNonQuery` в SQL Server достигает тысяч `INSERT` в секунду. В другом решении используются пакетные коммиты вместо отдельных команд `INSERT`, что позволяет избежать накладных расходов на протоколирование каждой команды `INSERT`.

Использование брокера сообщений

Чтобы справиться с выбросами трафика при записи, можно воспользоваться потоковым решением, например, поместив сервис Kafka перед сервисами SQL.

Одно из возможных решений показано на рис. 7.8. Когда автор отправляет новый или обновленный пост, хосты сервиса записи постов могут публиковать данные в топик Post. Сервис не имеет состояния и горизонтально масштабируем. Можно создать новый сервис, который мы назовем Post Writer; он непрерывно потребляет события из топика Post и записывает в сервис SQL. Сервис SQL может использовать репликацию по схеме «лидер/последователь», которая обсуждалась в главе 3.

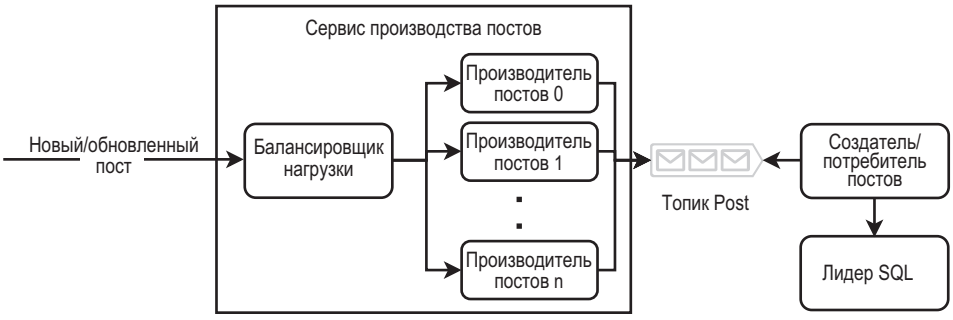

Рис. 7.8. Использование горизонтального масштабирования и брокера сообщений для обработки выбросов трафика записи

Главные недостатки этого подхода — сложность и согласованность в конечном счете. Вероятно, в вашей организации уже имеется сервис Kafka, которым вы можете воспользоваться; тогда вам не придется создавать собственный сервис, что отчасти снижает сложность. Интервал согласованности в конечном счете возрастает, так как операциям записи требуется больше времени для достижения последователей SQL.

Если требуемая пропускная способность записи превышает среднюю пропускную способность записи одного хоста SQL, можно продолжить функциональное секционирование кластеров SQL и создать выделенные кластеры SQL для категорий с интенсивным трафиком записи. Такое решение не идеально, поскольку логике приложения для просмотра постов придется читать из конкретных кластеров SQL в зависимости от категории. Логика запросов уже не инкапсулируется в сервисе SQL, но тоже присутствует в приложении. Сервис SQL перестал быть независимым от сервиса бэкенда, и обслуживание обоих сервисов усложняется.

Если вам нужна более высокая пропускная способность записи, вы можете использовать базу данных NoSQL, такую как Cassandra или Kafka с HDFS.

Также можно обсудить добавление ограничителя частоты запросов (см. главу 8) перед кластером бэкенда для предотвращения DDoS-атак.

7.14. СЕРВИС ЭЛЕКТРОННОЙ ПОЧТЫ

Бэкенд может выдавать к общему сервису электронной почты запросы на отправку сообщений. Чтобы отправить авторам напоминание о необходимости продлить пост, когда его возраст достигнет 7 дней, можно реализовать пакетное задание ETL, которое обращается к базе данных SQL с запросом постов возрастом 7 дней, а затем выдает к сервису электронной почты запросы на отправку сообщения для каждого поста.

Для сервиса оповещений в других приложениях могут быть такие требования, как обработка непредвиденных выбросов трафика, низкая задержка и доставка уведомлений за короткое время. Такой сервис уведомлений рассматривается в следующей главе.

7.15. ПОИСК

В разделе 2.6 мы создаем индекс Elasticsearch для таблицы Post, чтобы пользователи могли выполнять поиск постов. Можно обсудить, нужно ли давать пользователю возможность фильтровать посты до и после поиска (по пользователю, цене, состоянию, местоположению, возрасту поста и т. д.), и вносить соответствующие изменения в индекс.

7.16. УДАЛЕНИЕ СТАРЫХ ПОСТОВ

Посты Craigslist устаревают по прошествии определенного периода времени, после которого пост становится недоступным. Эта функциональность может быть реализована заданием cron или Airflow, ежедневно обращающимся к конечной точке `DELETE /old_posts`. `DELETE /old_posts` может быть самостоятельной конечной точкой, отдельной от `DELETE /post/{id}`, потому что последняя выполняет простую операцию удаления из базы данных, тогда как первая содержит более сложную логику для предварительного вычисления метки времени и удаления постов, возраст которых превышает вычисленное значение. Возможно, обе конечные точки также должны удалять соответствующие ключи из кэша Redis.

Это простая задача не критической важности, потому что если посты, которые должны быть удалены, останутся доступными в течение какого-то времени, ничего страшного не произойдет. Таким образом, простого задания cron может быть достаточно, а Airflow может внести лишнюю сложность. Также необходимо следить, чтобы не удалить случайно посты до их устаревания, так что любые изменения в этой функциональности должны тщательно тестироваться в промежуточной среде до их развертывания на продакшен. Преимущество cron перед сложными платформами управления рабочими процессами (такими, как Airflow) в упрощении обслуживания, особенно если инженер, разработавший функциональность, ушел из проекта и за обслуживание теперь отвечает другой инженер.

Преимущества удаления старых постов и старых данных вообще:

- Денежная экономия на предоставлении хранилищ и обслуживании.
- Ускорение операций с базами данных (таких, как чтение и индексирование).
- Операции обслуживания, требующие копирования всех данных в новое место, выполняются быстрее, проще и дешевле, чем добавление другого решения баз данных или миграция на него.

- Меньше проблем с безопасностью для организации и ограничение эффекта от несанкционированного доступа к данным, хотя это преимущество не такое сильное, поскольку это общедоступные данные.

Недостатки:

- Невозможность сбора аналитики и полезной информации, которую можно получить при хранении данных.
- Регуляторные органы могут требовать хранения данных в течение определенного времени.
- Существует ничтожная вероятность того, что URL-адрес удаленного поста может быть использован для нового поста, и читатель может подумать, что он просматривает старый пост. Вероятность таких событий повышается при использовании сервисов сокращения ссылок. Впрочем, вероятность этого события низка, и его эффект не очень велик, так что уровень риска приемлем: он станет неприемлемым, если существует угроза раскрытия конфиденциальных персональных данных.

Если затраты становятся проблемой, а обращения к старым данным редки, альтернативой удалению данных может быть сжатие, выполняемое при сохранении на дешевом оборудовании для архивации (например, магнитных лентах) или сетевом сервисе архивации данных, таком как AWS Glacier или Azure Archive Storage. Если возникнет необходимость в старых данных, их можно будет записать на диски до операций обработки данных.

7.17. МОНИТОРИНГ И ОПОВЕЩЕНИЯ

Кроме того, что обсуждалось в разделе 2.5, следует организовать мониторинг и отправку оповещений:

- Решение мониторинга базы данных (рассматривается в главе 10) должно инициировать сигнал низкой срочности, если старые посты не удаляются.
- Обнаружение аномалий:
  - количество добавленных или удаленных постов;
  - высокое количество поисков по конкретному термину;
  - количество постов, помеченных как неуместные.

7.18. ИТОГОВЫЙ ДИЗАЙН АРХИТЕКТУРЫ

На рис. 7.9 изображена архитектура Craigslist, включающая многие сервисы, которые мы обсуждали: клиент, бэкенд, SQL, кэш, сервис уведомлений, сервис поиска, хранилище объектов, CDN, ведение журналов, мониторинг, оповещения и пакетные ETL.

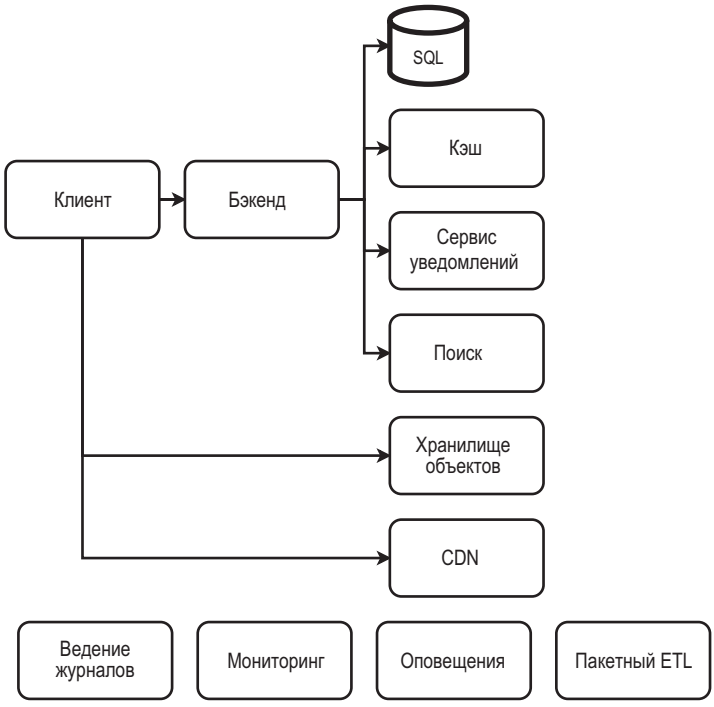

Рис. 7.9. Архитектура Craigslist с сервисом уведомлений, поиском, ведением журналов, мониторингом и оповещениями. Сервисы ведения журналов, мониторинга и оповещений могут обслуживать много других компонентов, поэтому на диаграмме они показаны как слабосвязанные компоненты. На пакетном сервисе ETL можно ставить задачи для таких целей, как периодическое удаление старых постов

7.19. ДРУГИЕ ВОЗМОЖНЫЕ ТЕМЫ ОБСУЖДЕНИЙ

Наш дизайн системы выполняет требования, перечисленные в начале главы. Оставшаяся часть собеседования может быть посвящена новым ограничениям и требованиям.

7.19.1. Жалобы на посты

Мы еще не обсуждали функциональность, позволяющую пользователям жаловаться на посты администратору, потому что она реализуется довольно просто. Ее обсуждение может включать дизайн системы для выполнения следующих требований:

- По получении определенного количества жалоб пост будет удален, а автор получит сообщение по электронной почте.

- Учетная запись и адрес электронной почты автора автоматически блокируются, так что автор не сможет входить в Craigslist или создавать посты. Тем не менее он может продолжить просмотр постов без ввода учетных данных и отправлять электронные письма другим авторам.
- Автор должен иметь возможность связаться с администратором, чтобы опротестовать это решение. Возможно, стоит обсудить с экспертом, понадобится ли специальная система для отслеживания и сохранения подобных взаимодействий и решений.
- Если автор захочет заблокировать сообщения электронной почты, ему придется настроить собственную учетную запись электронной почты для блокировки адреса отправителя. Craigslist такой возможности не дает.

7.19.2. Корректное снижение функциональности

Как обработать сбой в каждом компоненте? Каковы возможные граничные случаи, которые могут вызвать сбои, и как с ними справиться?

7.19.3. Сложность

Craigslist проектируется как простое приложение, оптимизированное для обслуживания небольшой группой инженеров. Функциональность намеренно ограничена и четко определена, и новые функции добавляются редко. Возможно, вы будете обсуждать стратегии для достижения этой цели.

Минимизация зависимостей

Любое приложение, содержащее зависимости от библиотек и/или сервисов, естественным образом деградирует со временем. Разработчикам приходится заниматься его обслуживанием только для того, чтобы оно продолжало поддерживать свою текущую функциональность. Старые версии библиотек, а со временем и целые библиотеки устаревают, а сервисы выходят из эксплуатации. Все это заставляет разработчиков устанавливать новые версии или искать альтернативы. Новые версии библиотек или развертывания сервисов могут нарушить работоспособность приложения. Необходимость в обновлении библиотек также может возникнуть при обнаружении ошибок или дефектов безопасности в текущих библиотеках. Минимизация функциональности системы минимизирует число ее зависимостей, что упрощает отладку, диагностику и обслуживание.

Такой подход требует соответствующей культуры, ориентированной на предоставление минимального набора функций, не требующих глубокой настройки для каждого рынка. Например, основная причина, по которой Craigslist не предоставляет функциональность платежей, заключается в том, что в разных городах может использоваться разная бизнес-логика обработки платежей.

Необходимо учитывать разные валюты, налоги, платежные операторы (MasterCard, Visa, PayPal, WePay и т. д.), и потребуется постоянная синхронизация изменений этих факторов. В культуре многих крупных технологических компаний принято вознаграждать руководителей программ и инженеров за концептуализацию и построение новых сервисов; в данном примере такая культура неуместна.

Использование облачных сервисов

На рис. 7.9 каждый сервис может быть развернут в облачном сервисе, кроме клиента и бэкенда. Например, можно использовать следующие сервисы AWS для каждого из сервисов на рис. 7.9. Другие облачные провайдеры (такие, как Azure или GCP) предоставляют аналогичные сервисы:

- SQL: RDS (https://aws.amazon.com/rds/)
- Хранилище объектов: S3 (https://aws.amazon.com/s3/)
- Кэш: ElastiCache (https://aws.amazon.com/elasticache/)
- CDN: CloudFront (https://www.amazonaws.cn/en/cloudfront/)
- Сервис уведомлений: Simple Notiication Service (https://aws.amazon.com/sns)
- Поиск: CloudSearch (https://aws.amazon.com/cloudsearch/)
- Ведение журналов, мониторинг и оповещения: CloudWatch (https://aws.amazon.com/cloudwatch/)
- Пакетный сервис ETL: лямбда-функции с выражениями rate или cron (https://docs.aws.amazon.com/lambda/latest/dg/services-cloudwatchevents-expressions.html)

Хранение целых веб-страниц в виде документов html

Веб-страница обычно состоит из шаблона HTML с чередованием функций JavaScript, которые отправляют запросы на бэкенд для заполнения значениями. В случае Craigslist шаблон поста в виде страницы HTML может содержать такие поля, как название, описание, цена, фото и т. д., а значение каждого поля заполняется в результате выполнения кода JavaScript.

Простой и компактный дизайн веб-страницы поста Craigslist делает возможной простую альтернативу, которая рассматривалась в разделе 7.5 и которую мы продолжим обсуждать здесь. Веб-страница поста может храниться в виде одного документа HTML в базе данных или CDN. Возможен такой простой вариант, как хранение пары «ключ — значение»: ключом является идентификатор поста, а значением — документ HTML. Решение жертвует частью хранилища данных, потому что в каждой записи базы данных будет присутствовать дублирующаяся разметка HTML. Для списка идентификаторов постов можно построить индексы поиска. Такой подход также упрощает добавление или удаление полей из новых постов. Если вы решите добавить новое обязательное поле (например,

подзаголовок), поля можно изменить без миграции базы данных SQL. Изменять поля в старых постах не нужно — их срок жизни ограничен, и эти сообщения будут автоматически удаляться. Таблица Post упрощается, поля поста заменяются его URL-адресом в CDN. Остаются столбцы `id`, `ts`, `poster_id`, `location_id`, `post_url`.

Наблюдаемость

Любое обсуждение обслуживания должно подчеркивать важность наблюдаемости, подробно рассмотренной в разделе 2.5. Необходимо потратиться на ведение журналов, мониторинг, оповещения, автоматизацию тестирования и внедрение хороших практик SRE, включая качественные дашборды для мониторинга, документацию и автоматизацию отладки.

7.19.4. Категории / теги постов

Для предложений Craigslist можно определить категории/теги — «автомобили», «недвижимость», «мебель» и т. д., чтобы автор мог закрепить за своим предложением определенное количество тегов (например, три). Для тегов создается отдельная таблица SQL. В таблицу Post включается столбец для списка тегов, разделенных запятыми. Альтернативное решение — создание соединительной таблицы post_tag, как показано на рис. 7.10.

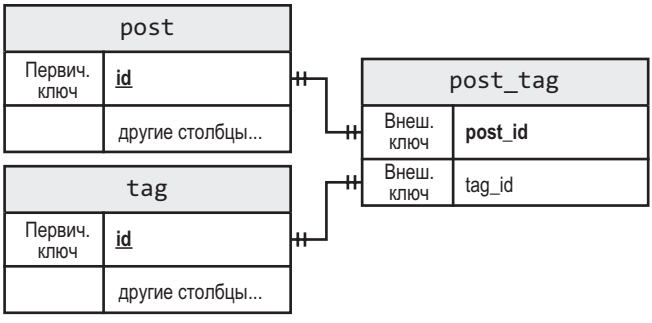

Рис. 7.10. Соединительная таблица для постов и тегов. Нормализация схемы обеспечивает согласованность за счет предотвращения дублирования данных. Если все данные хранятся в одной таблице, в строках появятся дублирующиеся значения

Плоский список можно расширить в иерархический список, чтобы пользователи могли применять более точные фильтры для просмотра постов, соответствующих их интересам. Например, категория «недвижимость» может иметь следующие вложенные подкатегории:

- Недвижимость > Тип операции > Аренда
- Недвижимость > Тип операции > Продажа

- Тип жилья > Квартира
- Тип жилья > Дом с одной спальней
- Тип жилья > Таунхаус

7.19.5. Аналитика и рекомендации

Можно создать ежедневные пакетные задания ETL, которые обращаются с запросами к базе данных SQL и заполняют дашборды разными метриками:

- Количество предложений по тегам.
- Теги, на которые приходится большинство щелчков.
- Теги с наибольшим количеством читателей, связывающихся с авторами.
- Теги с наибольшим количеством продаж, что можно измерить временем, по прошествии которого автор удалил пост после его публикации.
- Количество постов с жалобами, подозрительным и подтвержденным мошенническим статусом, их распределение по географии и времени.

Craigslist не поддерживает персонализацию, и посты упорядочиваются по новизне. На собеседовании можно обсудить возможность персонализации, которая включает отслеживание активности пользователей и механизм рекомендации постов.

7.19.6. A/B-тестирование

Как кратко упоминалось в разделе 1.4.5, в ходе разработки новой функциональности и визуального дизайна приложения можно постепенно развертывать их для все большего количества пользователей (а не для всех пользователей сразу).

7.19.7. Подписки и сохраненные поисковые запросы

Вы можете предоставить конечную точку API для сохранения критериев пользовательского поиска (с ограничением количества символов) и оповещения пользователей (по электронной почте, в мессенджерах, внутри приложения и т. д.) о любых новых постах, соответствующих сохраненным поисковым запросам. Запрос POST к этой конечной точке может записать строку данных (`timestamp`, `user_id`, `search_term`) в таблицу SQL, которую мы назовем saved_search. Сервис подписки на сохранение поисковых запросов сам по себе может быть довольно сложной системой, как будет показано в этом разделе.

Пользователь должен получать одно ежедневное уведомление, объединяющее все сохраненные запросы. Уведомление может содержать список критериев поиска и до десяти соответствующих результатов по каждому из них. В свою очередь, каждый результат состоит из списка данных постов для этих критериев. Данные

могут включать ссылку на пост и сводную информацию (название, цена, первые 100 символов описания) для отображения в уведомлении.

Например, если пользователь сохранил два критерия поиска, san francisco studio apartment и systems design interview book, уведомление может выглядеть так (конечно, вам не нужно записывать все это в ходе собеседования. Вы можете набросать несколько фрагментов и на словах объяснить, что они означают).

```
[
  {
    "search_term": "san francisco studio apartment",
    "results": [
      {
        "link": "sfbay.craigslist.org/12345",
        "title": "Totally remodeled studio",
        "price": 3000,
        "description_snippet": "Beautiful cozy studio apartment in the
     Mission. Nice views in a beautiful and safe neighborhood. Clo"
      },
      {
        "link": "sfbay.craigslist.org/67890"
        "title": "Large and beautiful studio",
        "price": 3500,
        "description_snippet": "Amenities\nComfortable, open floor plan\nIn
     unit laundry\nLarge closets\nPet friendly\nCeiling fan\nGar"
      },
      ...
    ]
  },
  {
    "search_term": "systems design interview book",
    "results": [
      ...
    ]
  }
]
```

Для отправки пользователям новых результатов по их сохраненным запросам можно реализовать ежедневное пакетное задание ETL. Существует как минимум два способа реализации этого задания: простой, допускающий дубликаты запросов к сервису поиска, и более сложный, без отправки дублирующихся запросов.

7.19.8. Решение с дубликатами запросов к сервису поиска

Elasticsearch кэширует частые запросы поиска (https://www.elastic.co/blog/elasticsearch-caching-deep-dive-boosting-query-speed-one-cache-at-a-time), чтобы частые запросы с одинаковыми условиями поиска не расходовали много ресурсов. Пакетное задание ETL может обрабатывать пользователей и их сохраненные условия поиска по одному. Каждый процесс включает отправку условий поиска

пользователя в отдельных запросах сервису поиска, консолидацию результатов и последующую отправку запроса сервису уведомлений (тема главы 9).

7.19.9. Решение без дубликатов запросов к сервису поиска

Пакетное задание ETL выполняет следующие действия:

1. Дедупликация условий поиска, чтобы поиск выполнялся для каждого условия только один раз. Для дедупликации вчерашних условий поиска можно выполнить запрос SQL вида `SELECT DISTINCT LOWER(search_term) FROM saved_search WHERE timestamp >= UNIX_TIMESTAMP(DATEADD(CURDATE(), INTERVAL -1 DAY)) AND timestamp < UNIX_TIMESTAMP(CURDATE())`. Поиск может выполняться без учета регистра, поэтому условие поиска в процессе дедупликации преобразуется к нижнему регистру. Так как архитектура Craigslist секционирована по городам, количество условий поиска не должно превысить 100 М. Если предположить, что условие поиска состоит в среднем из 10 символов, объем данных составит 1 Гбайт, то есть легко поместится в памяти одного хоста.
2. Для каждого условия поиска:
   a) Отправить запрос к сервису поиска (Elasticsearch) и получить результаты.
   b) Запросить из таблицы saved_search идентификатор пользователя, связанный с этим условием поиска.
   c) Для каждого кортежа (идентификатор пользователя, условие поиска, результаты) отправить запрос сервису уведомлений.

Что, если произойдет сбой на шаге 2? Как предотвратить повторную отправку уведомлений пользователям? Можно воспользоваться механизмом распределенных транзакций, описанным в главе 5. А можно реализовать в клиенте логику, которая, прежде чем вывести уведомление, проверяет, было ли оно уже показано (и возможно, закрыто). Это осуществимо для некоторых типов клиентов (например, браузеров или мобильных приложений), но не для электронной почты или текстовых сообщений.

Если срок действия сохраненных поисковых запросов истекает, можно удалять старые данные ежедневным потоковым заданием, которое выполняет команду SQL `DELETE` для строк с возрастом, превышающим срок действия.

7.19.10. Ограничение частоты запросов

Все запросы к сервису могут проходить через ограничитель частоты, чтобы отдельные пользователи не могли отправлять запросы слишком часто, тем самым

потребляя слишком много ресурсов. Структура ограничителя частоты описана в главе 8.

7.19.11. Большее количество постов

Что, если предоставить единый URL, по которому любой пользователь (независимо от его местоположения) сможет увидеть все предложения? Тогда таблица Post может стать слишком большой для одного хоста, и индекс Elasticsearch для постов тоже может быть слишком большим для одного хоста. Но обслуживание запросов поиска должно производиться с одного хоста. Любой дизайн, в котором запрос обрабатывается несколькими хостами, а результаты агрегируются на одном хосте перед их возвращением читателю, будет иметь более высокую задержку и затраты. Как продолжить обслуживание запросов поиска с одного хоста? Варианты:

- Установить срок действия постов (период удержания) в 1 неделю и реализовать ежедневное пакетное задание для удаления постов с истекшим сроком. Короткий период удержания означает сокращение объема данных для поиска и кэширования, с уменьшением затрат и сложности системы.
- Сократить объем данных, хранимых в посте.
- Выполнить функциональное секционирование по категориям постов. Возможно, стоит создать отдельные таблицы SQL для разных категорий. Но тогда в приложении должны храниться данные сопоставления таблиц. Или сопоставление может храниться в кэше Redis, а приложение должно будет обращаться к нему, чтобы определить, к какой таблице обратиться.
- Сжатие рассматривать не стоит, потому что поиск в сжатых данных чересчур затратный.

7.19.12. Местные правила и нормы

В каждой юрисдикции (стране, штате, округе, городе и т. д.) могут действовать собственные правила и нормы, влияющие на работу Craigslist. Несколько примеров:

- Типы товаров или сервисов, разрешенных на Craigslist, могут отличаться в зависимости от юрисдикции. Как системе обработать это требование? Возможное решение обсуждается в разделе 15.10.1.
- Нормы, относящиеся к данным клиентов и конфиденциальности, могут запретить компании выводить данные за пределы страны. Возможно, вам придется обеспечить возможность удаления данных клиентов по их требованию или передачу данных правительственным организациям. Такие темы обычно не рассматриваются на собеседованиях.

Вы должны обсудить точные требования. Достаточно ли выборочно показывать в клиентском приложении разделы с товарами и услугами в зависимости от местонахождения пользователя или же следует запретить пользователям просмотр или публикацию объявлений о запрещенных товарах и услугах?

В исходном варианте выборочного показа разделов в клиенты добавляется логика вывода или скрытия разделов на основании страны, связанной с IP-адресом пользователя. Если же регулирующие нормы многочисленны или часто изменяются, возможно, придется создать нормативный сервис, который администраторы Craigslist будут использовать для настройки законодательных требований, а клиенты будут обращаться с запросами к этому сервису, чтобы определять, какую разметку HTML выводить или скрывать. Так как трафик чтения в сервисе будет намного интенсивнее, чем трафик записи, можно применить методы CQRS для обеспечения успешной записи. Например, можно создать разные нормативные сервисы для администраторов и читателей, которые масштабируются по отдельности, и периодически синхронизировать их.

Если необходимо запретить публикацию в Craigslist определенного контента, можно обсудить систему поиска запрещенных слов или выражений или даже методы машинного обучения.

Остается заметить, что Craigslist не настраивает предложения по странам. Хороший пример — удаление раздела знакомств в 2018 году в ответ на новые законодательные акты США. Сервис не стал сохранять этот раздел в других странах. Можно обсудить достоинства и недостатки такого подхода.

ИТОГИ

- Для определения нефункциональных требований, которыми в системе Craigslist являются масштабируемость, высокая доступность и высокая производительность, следует изучить пользовательские истории и необходимые типы данных (текст, изображения, видео и т. д.).
- CDN — стандартное решение для поставки изображения или видео, но не стоит полагать, что оно всегда самое подходящее. Используйте хранилище объектов, если медиаматериалы будут востребованы небольшой частью пользователей.
- Обсуждение масштабирования стоит начать с функционального секционирования с использованием GeoDNS.
- Далее нужно рассмотреть кэширование и CDN, в основном для повышения масштабируемости и снижения задержки предоставления постов.
- В нашем сервисе Craigslist испытывает большую нагрузку на чтение. Если вы используете SQL, рассмотрите репликацию по схеме «лидер — последователь» для масштабирования чтения.

- Используйте горизонтальное масштабирование бэкенда и брокеров сообщений для обработки выбросов трафика записи. Такая конфигурация обслуживает запросы записи, распределяя их между хостами бэкенда, и буферизует их в брокере сообщений. Потребительский кластер потребляет запросы от брокера сообщений и обрабатывает их соответствующим образом.
- Используйте пакетные или потоковые задания для любой функциональности, не требующей задержки реального времени. Этот вариант медленнее, но у него лучше масштабируемость и ниже затраты.
- Оставшаяся часть собеседования может быть посвящена новым ограничениям и требованиям. В этой главе среди них упоминались возможности отправки жалоб на посты, корректное сокращение функциональности, уменьшение сложности, добавление категорий/тегов к постам, аналитика и рекомендации, A/B-тестирование, подписки и сохраненные поисковые запросы, ограничение частоты, выдача большего количества постов каждому пользователю и соблюдение местных правил и норм.

8 Проектирование сервиса для ограничения частоты запросов

В ЭТОЙ ГЛАВЕ

- ✓ Ограничение частоты запросов
- ✓ Обсуждение сервиса для ограничения частоты запросов
- ✓ Различные алгоритмы ограничения частоты запросов

Ограничение частоты запросов — распространенный сервис, о котором почти всегда заходит речь на собеседованиях по проектированию систем и который встречается во многих примерах этой книги. В этой главе мы рассмотрим два сценария: 1) когда вы упомянете об ограничении частоты запросов, эксперт попросит вас раскрыть эту тему подробнее; 2) вам предложат спроектировать сервис ограничения частоты запросов.

Ограничение частоты определяет частоту, с которой потребители могут выдавать запросы к конечным точкам API. Ограничение частоты предотвращает случайное или злонамеренное создание повышенной нагрузки, особенно со стороны ботов. В этой главе мы будем называть такие клиенты «избыточными» (excessive clients).

Примеры непреднамеренной перегрузки:

- Клиентом является другой веб-сервис, который столкнулся с выбросом трафика (законным или злонамеренным).

- Разработчики этого сервиса решили провести нагрузочное тестирование на продакшен.

Такая непреднамеренная перегрузка порождает проблему «шумного соседа», когда клиент расходует слишком много ресурсов вашего сервиса, так что другие клиенты сталкиваются с более высокой задержкой или более высокой частотой запросов со сбоем.

Возможны и злонамеренные действия, включая перечисленные ниже (некоторые виды атак ботов, которые не предотвращаются ограничением частоты, подробности см. на странице https://www.cloudlare.com/learning/bots/what-is-bot-management/).

- *Атаки отказа в обслуживании*, или *DoS* (Denial-of-Service), или *распределенные атаки отказа в обслуживании*, DDoS (Distributed Denial-of-Service) — DoS-атака переполняет цель запросами, из-за чего обработка трафика становится невозможной. DoS использует для атаки одну машину, а DDoS — несколько машин. В этой главе данное различие несущественно, и мы будем обозначать их общим термином DoS.
- *Атаки методом «грубой силы» (брутфорса)* — атаки этой категории представляют собой многократные попытки подбора конфиденциальных данных (паролей, ключей шифрования, ключей API и регистрационных данных входа SSH) методом проб и ошибок.
- *Веб-скрейпинг,* или автоматическое извлечение данных, — боты используются для отправки запросов GET ко многим веб-страницам веб-приложения и получения большого объема данных. Пример — автоматическое извлечение данных о ценах и продуктах со страниц Amazon.

Ограничение частоты запросов может быть реализовано в виде библиотеки или в виде отдельного сервиса, вызываемого фронтендом, шлюзом API или сервисной сетью. Мы реализуем его в форме сервиса, чтобы получить преимущества функционального секционирования, описанного в главе 6. На рис. 8.1 изображена схема ограничителя частоты, которая будет рассматриваться в этой главе.

8.1. АЛЬТЕРНАТИВЫ СЕРВИСА ДЛЯ ОГРАНИЧЕНИЯ ЧАСТОТЫ ЗАПРОСОВ И ПОЧЕМУ ОНИ НЕРЕАЛИЗУЕМЫ

Почему бы не масштабировать сервис, организовав мониторинг нагрузки и добавляя новые хосты по мере необходимости вместо ограничения частоты запросов? Можно спроектировать сервис так, чтобы он был горизонтально масштабируемым и в него можно было добавить новые хосты для обслуживания дополнительной нагрузки. Можно рассмотреть возможность автоматического масштабирования.

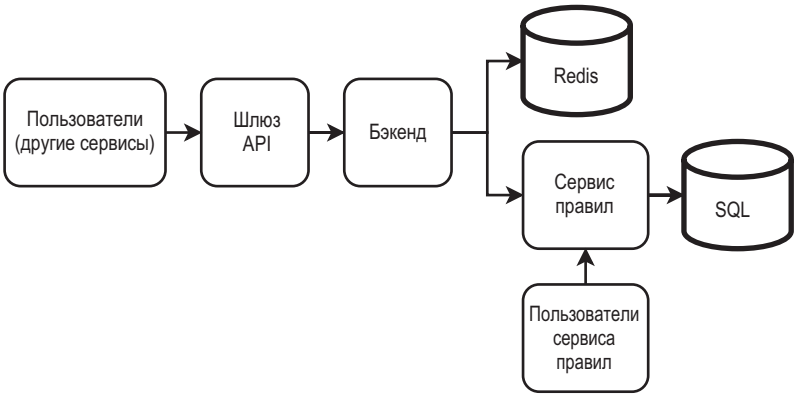

Рис. 8.1. Первоначальная высокоуровневая архитектура ограничителя частоты запросов. Сервисы фронтенда, бэкенда и правил также ведут протоколирование через общий сервис ведения журналов, не показанный на схеме. База данных Redis обычно реализуется в виде общего сервиса Redis (вместо предоставления собственной базы данных Redis нашим сервисом). Пользователи сервиса правил могут выдавать запросы API к сервису правил через браузерное приложение. Правила могут храниться в базе данных SQL

Процесс добавления новых хостов при обнаружении выброса трафика может быть слишком медленным. Добавление нового хоста включает ряд шагов, требующих времени: например, предоставление оборудования хоста, загрузку необходимых контейнеров Docker, запуск сервиса на новом хосте и последующее обновление конфигурации балансировщика нагрузки для направления трафика на новый хост. Процесс может быть слишком медленным, и к тому моменту, когда новые хосты будут готовы обслуживать трафик, сервис уже может выйти из строя. Даже решения автоматического масштабирования могут быть слишком медленными.

Балансировщик нагрузки может ограничить количество запросов, отправляемых каждому хосту. Почему бы не воспользоваться балансировщиком нагрузки, чтобы предотвратить перегрузку хостов и потерю запросов, когда в кластере не остается свободных мощностей?

Как уже отмечалось, система не должна обслуживать вредоносные запросы. Чтобы защитить ее от таких запросов, ограничитель частоты определяет их IP-адреса и просто игнорирует их. Как будет показано ниже, ограничитель частоты обычно возвращает код 429 («Слишком много запросов»), но если вы уверены, что часть запросов являются вредоносными, можно выбрать один из следующих вариантов:

- отклонить запрос и не возвращать никакого ответа, чтобы атакующий думал, что сервис утратил работоспособность;
- теневая блокировка пользователя — возврат кода 200 с пустым или дезинформирующим ответом.

Почему ограничение частоты необходимо выделять в отдельный сервис? Почему бы хостам независимо друг от друга не отслеживать частоту запросов от источников и ограничивать их по частоте?

Дело в том, что какие-то запросы обходятся дороже других. Некоторые пользователи отправляют запросы, возвращающие больше данных, требующие более дорогостоящей фильтрации и агрегирования или включающие операции JOIN между большими наборами данных. Обработка затратных запросов от некоторых клиентов может замедлить работу хоста.

Балансировщик нагрузки 4-го уровня не может обрабатывать содержимое запроса. Для липких сеансов (для маршрутизации запросов от пользователя к одному хосту) понадобится балансировщик нагрузки 7-го уровня, что повышает затраты и сложность. Если других причин использовать его нет, возможно, его применение для одной-единственной цели неоправданно и лучше подойдет специализированный общий сервис ограничения частоты запросов. Таблица 8.1 обобщает сказанное выше.

Таблица 8.1. Сравнение ограничения частоты запросов с альтернативами

Ограничение частоты запросов	Добавление новых хостов	Распределитель нагрузки 7-го уровня
Обрабатывает выбросы трафика, возвращая код 429 (Слишком много запросов) пользователям с высокой частотой запросов	Добавление новых хостов может быть слишком медленным, чтобы отреагировать на выбросы трафика. К тому моменту, когда новые хосты будут готовы к обслуживанию трафика, сервис может выйти из строя	Не решает проблему обработки выбросов трафика
Решает проблему DoS-атак возвращением дезинформирующих ответов	Обрабатывает вредоносные запросы, чего делать не следует	Не решает проблему
Может ограничивать по частоте пользователей, отправляющих затратные запросы	Вынуждает сервис нести затраты на обработку затратных запросов	Отклоняет затратные запросы, но может быть слишком сложным и затратным как самостоятельное решение

8.2. КОГДА НЕ СЛЕДУЕТ ПРИМЕНЯТЬ ОГРАНИЧЕНИЕ ЧАСТОТЫ ЗАПРОСОВ

Ограничение частоты подходит не для любых видов перегрузки со стороны клиента. Возьмем, к примеру, спроектированный нами сервис социальной сети. Пользователь может подписаться на обновления, связанные с конкретным хештегом. Если пользователь отправляет слишком много запросов на подписку

в течение определенного периода времени, сервис социальной сети может ответить: «Вы отправили слишком много запросов на подписку за последние несколько минут». В случае ограничения частоты запросов вы просто заблокируете запросы пользователя и вернете код 429 («Слишком много запросов») или не вернете ничего, а клиент решит, что возвращен ответ 500. Это не лучший сценарий взаимодействия с пользователем. Если запрос отправляется браузером или мобильным приложением, приложение может сообщить клиенту о чрезмерном количестве отправленных запросов — такое решение лучше.

Другой пример — сервисы, которые взимают абонентскую плату в зависимости от частоты запросов (например, разный размер платы за 1000 или 10 000 запросов в час). Если клиент превышает квоту за определенный период времени, дальнейшие запросы не будут обрабатываться до наступления следующего периода. Общий сервис ограничения частоты не подходит, чтобы предупредить превышение квоты клиентами. Как обсуждается далее, общий сервис ограничения частоты хорошо работает только в простых ситуациях, но не подойдет для сложных — скажем, с назначением каждому клиенту собственного ограничения частоты.

8.3. ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

Наш сервис ограничения частоты является общим, а его пользователи — это в основном сервисы, используемые внешними, а не внутренними сторонами (например, сотрудниками). Мы будем называть такие сервисы «пользовательскими». Пользователь должен иметь возможность установить максимальную частоту запросов, после которой запросы от того же источника будут откладываться или отклоняться с кодом 429. За единицу времени можно выбрать интервал продолжительностью 10 или 60 секунд. Можно установить ограничение в 10 запросов за 10 секунд. Другие функциональные требования:

- Предполагается, что каждый пользовательский сервис должен ограничить частоту запросов своих источников между хостами, но ограничение частоты для пользователя не должно распространяться на все сервисы. Ограничение частоты устанавливается для каждого пользовательского сервиса отдельно.
- Пользователь может установить несколько ограничений частоты, по одному для каждой конечной точки. Более сложные конфигурации уровня пользователя (например, разные ограничения частоты для определенных источников/пользователей) не нужны. Ограничитель частоты должен быть низкозатратным и масштабируемым сервисом, понятным и простым в использовании.
- Пользователи должны иметь доступ к информации о том, какие пользователи ограничены по частоте, а также к меткам времени начала и завершения событий ограничения частоты. Мы предоставим для этого конечную точку.
- Можно обсудить с экспертом, нужно ли регистрировать каждый запрос, поскольку это потребует большого (и как следствие, дорогостоящего) объема

хранилища. Предположим, что это необходимо, и обсудим методы экономии места для сокращения затрат.

- Следует регистрировать источники запросов, ограниченные по частоте, для последующего ручного анализа. Особенно это касается подозрительной активности, за которой могут скрываться атаки.

8.4. НЕФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

Ограничение частоты — базовая функциональность, необходимая практически любому сервису. Она должна быть масштабируемой, иметь высокую производительность и быть максимально простой, безопасной и конфиденциальной. Ограничение частоты несущественно для доступности сервиса, так что вы можете поискать баланс между высокой доступностью и отказоустойчивостью. Точность и консистентность весьма важны, но не критичны.

8.4.1. Масштабируемость

Сервис должен масштабироваться до миллиардов ежедневных запросов информации о том, следует ли ограничивать по частоте конкретный источник запросов. Запросы на изменение ограничений частоты будут отправлять вручную внутренние пользователи организации, и предоставлять эту функциональность внешним пользователям не нужно.

Сколько памяти потребуется? Допустим, у сервиса миллиард пользователей, и в любой момент времени нужно хранить 100 запросов на пользователя. Хранить достаточно только идентификаторы пользователей и очереди из 100 меток времени на пользователя; каждая занимает 64 бита. Ограничитель частоты является общим сервисом, поэтому необходимо связать запросы с сервисом, ограничиваемым по частоте. Типичная крупная организация использует тысячи сервисов. Предположим, что из них ста сервисам требуется ограничение частоты.

Спросите себя, действительно ли ограничителю частоты необходимо хранить данные одного миллиарда пользователей. Какова продолжительность периода удержания? Ограничителю частоты обычно достаточно хранить данные всего 10 секунд, поскольку он принимает решение на основании частоты запросов пользователя за последние 10 секунд. Более того, вы можете обсудить с экспертом, насколько вероятно появление более 1–10 миллионов пользователей в пределах 10-секундного окна. Возьмем консервативную оценку в 10 миллионов пользователей. Общие требования к системе хранения составляют 100 * 64 * 101 * 10 Мбайт = 808 Гбайт. Если вы используете Redis и назначаете ключ каждому пользователю, размер значения составит 64 * 100 = 800 байт. Может быть нецелесообразно удалять данные сразу же после того, как они станут старше 10 секунд, так что фактический требуемый объем хранилища зависит от того, насколько быстро сервис может удалять старые данные.

8.4.2. Производительность

Когда другой сервис получает запрос от пользователя (мы называем такие запросы *пользовательскими*), он отправляет запрос к сервису ограничения частоты (в дальнейшем такие запросы будут называться *запросами ограничителя частоты*), чтобы определить, следует ли ограничить частоту запроса. Запрос ограничителя частоты является блокирующим; другой сервис не может ответить пользователю, пока не будет завершен запрос ограничителя. Время ответа на запрос ограничителя частоты добавляется к времени ответа на пользовательский запрос. Следовательно, сервис должен иметь очень низкую задержку — вероятно, 100 миллисекунд для 99-го процентиля. Решение о применении или неприменении ограничения частоты должно быть быстрым. Для просмотра или аналитики журналов низкая задержка не требуется.

8.4.3. Сложность

Сервис должен быть общим, совместно используемым другими сервисами организации. Его дизайн должен быть простым, чтобы минимизировать риск ошибок и неработоспособности, упростить диагностику, сосредоточиться на единственной функциональности ограничения частоты и минимизировать затраты. Интеграция решения для ограничения частоты с другими сервисами должна быть максимально простой и удобной.

8.4.4. Безопасность и конфиденциальность

В главе 2 рассматривались ожидания с точки зрения безопасности и конфиденциальности для внешних и внутренних сервисов. В этом разделе мы обсудим некоторые риски безопасности и конфиденциальности. Реализаций безопасности и конфиденциальности пользовательских сервисов может быть недостаточно для предотвращения внешних атак на сервис ограничения частоты. Внутренние пользовательские сервисы также могут пытаться атаковать ограничитель частоты, например, фальсифицируя запросы от другого пользовательского сервиса для ограничения его частоты. Пользовательские сервисы также могут нарушать конфиденциальность, запрашивая данные об источниках ограничения частоты у других пользовательских сервисов.

По этим причинам мы реализуем безопасность и конфиденциальность в дизайне системы ограничителя частоты.

8.4.5. Доступность и отказоустойчивость

Высокая доступность или отказоустойчивость могут не входить в список обязательных требований. Если доступность сервиса находится на уровне менее «трех девяток» и он неработоспособен в среднем на несколько минут ежедневно, в этот период пользовательские сервисы могут просто обрабатывать все запросы,

не соблюдая ограничение частоты. Более того, вместе с доступностью увеличиваются и затраты. Обеспечение 99,9% доступности обходится относительно недорого, тогда как уровень 99,99999% может оказаться неприемлемо дорогим.

Сервис можно спроектировать с использованием простого кэша с высокой доступностью для кэширования IP-адресов избыточных клиентов. Если сервис ограничения частоты запросов идентифицирует избыточных клиентов до сбоя, этот кэш может продолжить обслуживать запросы ограничителей частоты во время сбоя, так что для избыточных клиентов ограничение по частоте сохранится. Статистически маловероятно, что избыточный клиент появится за те несколько минут, пока сервис ограничения частоты неработоспособен. Если это все же произойдет, можно использовать другие методы (например, брандмауэры) для предотвращения сбоя сервиса за счет ухудшения опыта пользователя на эти несколько минут.

8.4.6. Точность

Чтобы не ухудшать пользовательский опыт, необходимо избежать ошибок определения избыточных клиентов и ограничения их по частоте. Если вы не уверены, не применяйте ограничение. Значения для ограничения не обязаны быть абсолютно точными. Например, если оно составляет 10 запросов за 10 секунд, иногда допустимо отклонение до 8 или 12 запросов за 10 секунд. Если действующее соглашение SLA требует обеспечить минимальную частоту запросов, можно установить более высокое ограничение (например, 12+ запросов за 10 секунд).

8.4.7. Консистентность

Точность ведет за собой консистентность. В нашем случае сильная консистентность не требуется. Когда пользовательский сервис обновляет ограничение частоты, новое ограничение не обязательно немедленно применять к новым запросам; несколько секунд несогласованности допустимы. Консистентность в конечном счете также приемлема для просмотра зарегистрированных событий — например, какие пользователи были ограничены по частоте — или аналитики журналов. Консистентность в конечном счете в отличие от сильной консистентности позволяет сделать дизайн более простым и экономичным.

8.5. ПОЛЬЗОВАТЕЛЬСКИЕ ИСТОРИИ И НЕОБХОДИМЫЕ КОМПОНЕНТЫ СЕРВИСА

Запрос ограничителя частоты содержит обязательный идентификатор пользователя и идентификатор пользовательского сервиса. Так как ограничение частоты определяется независимо для каждого пользовательского сервиса, формат идентификатора может быть разным для каждого сервиса. Этот формат

определяется и поддерживается пользовательским сервисом, а не сервисом ограничения частоты. Идентификатор пользовательского сервиса используется, чтобы различать возможные совпадающие идентификаторы пользователей в разных пользовательских сервисах. Так как каждый такой сервис имеет свое ограничение частоты, ограничитель частоты использует идентификатор для определения того, какое значение порога частоты применять.

Наш ограничитель частоты должен хранить данные (идентификатор пользователя, идентификатор сервиса) в течение 60 секунд, поскольку они необходимы для вычисления частоты запросов пользователя, чтобы определить, не превышает ли она пороговое значение. Чтобы свести к минимуму задержку получения частоты запросов пользователя или ограничение частоты любого сервиса, эти данные можно хранить (или кэшировать) в хранилище в памяти. Так как согласованность и задержка для журналов не важны, можно хранить журналы в хранилище, обеспечивающем согласованность в конечном счете (например, HDFS). В нем реализуется репликация для предотвращения потери данных из-за возможных сбоев хостов.

Наконец, пользовательские сервисы могут иногда отправлять запросы к сервису ограничения частоты, чтобы создавать и обновлять пороги частоты для своих конечных точек. Такой запрос может состоять из идентификатора пользовательского сервиса, идентификатора конечной точки и требуемого порога частоты (например, максимум 10 запросов за 10 секунд).

Объединяя все требования, получаем следующий список:

- База данных с быстрым чтением и записью для счетчиков. Схема будет простой, не сложнее простой пары (идентификатор пользователя, идентификатор сервиса). Можно воспользоваться базой данных в памяти, например, Redis.
- Сервис, в котором можно определять и получать правила; будем называть его сервисом правил.
- Сервис, который отправляет запросы к сервису правил и базе данных Redis; будем называть его сервисом бэкенда.

Два сервиса существуют отдельно друг от друга, потому что запросы к сервису правил для добавления или изменения правил не должны взаимодействовать с запросами к ограничителю частоты, который определяет, должен ли запрос подчиняться ограничению частоты.

8.6. ВЫСОКОУРОВНЕВАЯ АРХИТЕКТУРА

На рис. 8.2 (повторение рис. 8.1) изображена высокоуровневая архитектура в контексте рассмотренных требований и сценариев. Запрос от клиента к сервису ограничения частоты сначала проходит через фронтенд или сервисную сеть.

Если механизмы безопасности фронтенда пропускают запрос, он поступает на бэкенд, где выполняются следующие действия:

1. Получение порога частоты сервиса от сервиса правил. Значение может кэшироваться для снижения задержки и объема запросов к сервису правил.
2. Определение текущей частоты запросов сервиса, включая этот запрос.
3. Возвращение ответа, который сообщает, следует ли ограничить частоту запроса.

Шаги 1 и 2 могут выполняться параллельно для сокращения общей задержки. С этой целью для каждого шага выделяется отдельный программный поток или же используются потоки из общего пула.

Сервисы фронтенда и Redis (распределенный кэш) в высокоуровневой архитектуре на рис. 8.2 предназначены для горизонтальной масштабируемости. Это решение с распределенным кэшем, которое обсуждалось в разделе 3.5.3.

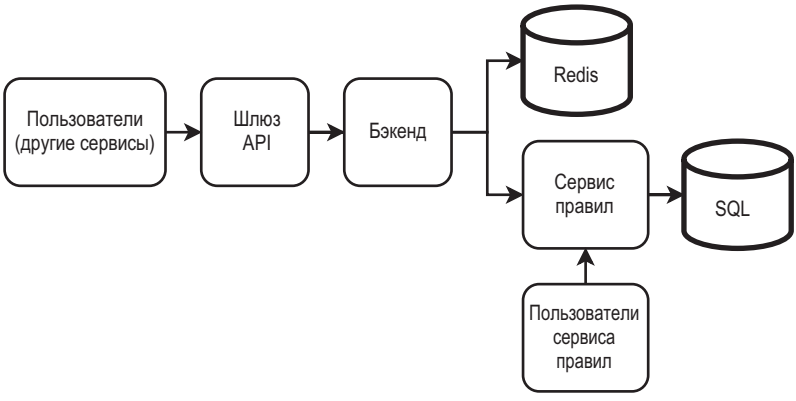

Рис. 8.2. Исходная высокоуровневая архитектура ограничителя частоты. Сервисы фронтенда, бэкенда и правил также ведут запись в журнал через общий сервис ведения журналов, не показанный на схеме. База данных Redis обычно реализуется в виде общего сервиса Redis (вместо собственной базы данных Redis для нашего сервиса). Пользователи сервиса правил могут отправлять запросы API к сервису правил через браузерное приложение

На рис. 8.2 можно заметить, что сервис правил получает запросы от пользователей двух разных сервисов (бэкенда и правил) с сильно различающимися объемами запросов, один из которых (сервис правил) выполняет всю запись.

Вспомним, что такое репликация «лидер — последователь» из разделов 3.3.2 и 3.3.3, представленная на рис. 8.3: пользователи сервиса правил могут отправлять все свои запросы SQL (как чтения, так и записи) узлу-лидеру. Бэкенд должен отправлять свои запросы SQL (только запросы на чтение / SELECT)

узлам-последователям. При таком подходе обеспечивается высокая консистентность и производительность сервиса правил.

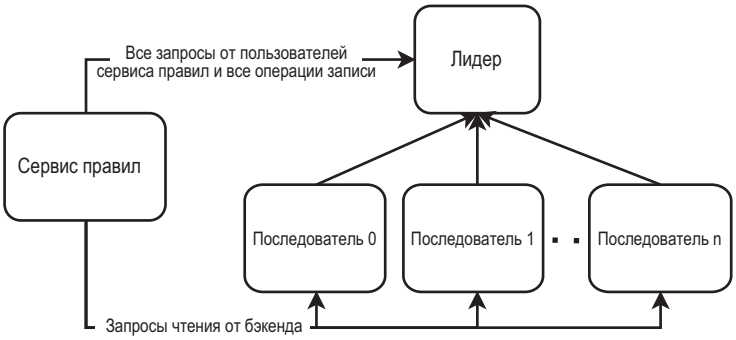

Рис. 8.3. Хост-лидер должен обрабатывать все запросы от пользователей сервиса правил и все операции записи. Операции чтения от бэкенда распределяются между хостами-последователями

Так как правила не должны изменяться часто, на рис. 8.4 можно добавить кэш Redis к сервису правил, чтобы дополнительно повысить производительность чтения. На рис. 8.4 показано кэширование с расширением, но можно использовать и другие стратегии кэширования из раздела 3.8. Сервис бэкенда также может кэшировать правила в Redis. Как упоминалось в разделе 8.4.5, можно кроме того кэшировать идентификаторы избыточных пользователей. Когда пользователь превышает свой порог частоты, его идентификатор кэшируется вместе со временем, после которого ограничение перестает действовать. Тогда бэкенду не придется обращаться с запросом к сервису правил для отклонения запроса пользователя.

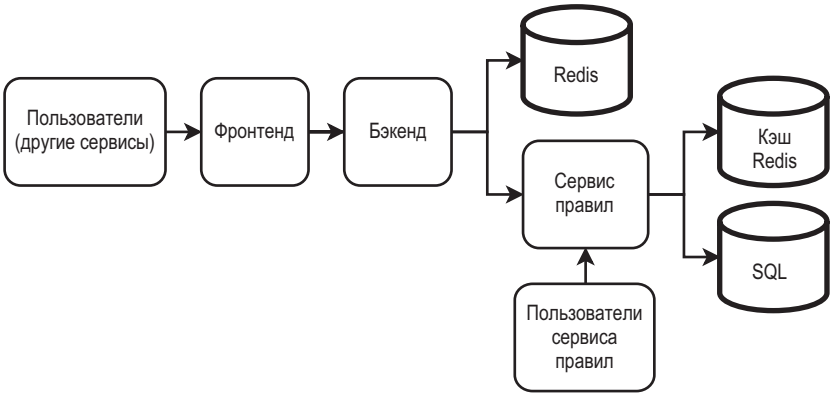

Рис. 8.4. Ограничитель частоты с кэшем Redis в сервисе правил. Частые запросы от бэкенда могут обслуживаться из кэша, а не из базы данных SQL

Если вы используете AWS (Amazon Web Services), рассмотрите DynamoDB вместо Redis и SQL. DynamoDB может обрабатывать миллионы запросов в секунду (https://aws.amazon.com/dynamodb/) с согласованностью в конечном счете или сильной согласованностью (https://docs.aws.amazon.com/whitepapers/latest/comparing-dynamodb-and-hbase-for-nosql/consistency-model.html), однако использование DynamoDB создает привязку к конкретной технологии.

Бэкенд соответствует всем нефункциональным требованиям. Он масштабируем, обладает высокой производительностью, несложен, безопасен, обеспечивает конфиденциальность и консистентность в конечном счете. База данных SQL с репликацией «лидер — лидер» обладает высокой доступностью и отказоустойчивостью, что выходит за рамки наших требований. Точность мы рассмотрим чуть позже. Этот дизайн не масштабируется для пользователей сервиса правил, что допустимо, как объяснялось в разделе 8.4.1.

С учетом наших требований исходная архитектура может оказаться переусложненной и слишком затратной. Этот дизайн обладает высокой точностью и сильной консистентностью, причем обе характеристики не входят в нефункциональные требования. Можно ли пожертвовать некоторой точностью и консистентностью для снижения затрат? Для начала рассмотрим два подхода к масштабированию ограничителя частоты:

1. Хост может обслуживать любого пользователя, при этом он не хранит состояние и загружает данные из общей базы данных. Это подход без сохранения состояния, который применялся в большинстве задач этой книги.
2. Хост обслуживает фиксированный набор пользователей и хранит данные своих пользователей. Это подход с состоянием, который будет рассмотрен в следующем разделе.

8.7. РЕШЕНИЕ С СОХРАНЕНИЕМ СОСТОЯНИЯ/ ШАРДИРОВАНИЕ

На рис. 8.5 изображен бэкенд решения с сохранением состояния, которое лучше соответствует нефункциональным требованиям. При поступлении запроса балансировщик нагрузки маршрутизирует его к хосту. Каждый хост хранит количество своих клиентов в памяти. Хост определяет, не превысил ли пользователь свой порог частоты, и возвращает `true` или `false`. Если пользователь отправляет запрос, а его хост неработоспособен, сервис возвращает ошибку 500 и запрос не будет ограничиваться по частоте.

Решение с сохранением состояния требует балансировщика нагрузки 7-го уровня. Может показаться, что это противоречит сказанному в разделе 8.1 об использовании такого балансировщика, однако сейчас мы обсуждаем его использование в распределенном решении ограничения частоты, а не только для липких сеансов,

чтобы каждый хост мог отклонять затратные запросы и выполнять ограничение частоты самостоятельно.

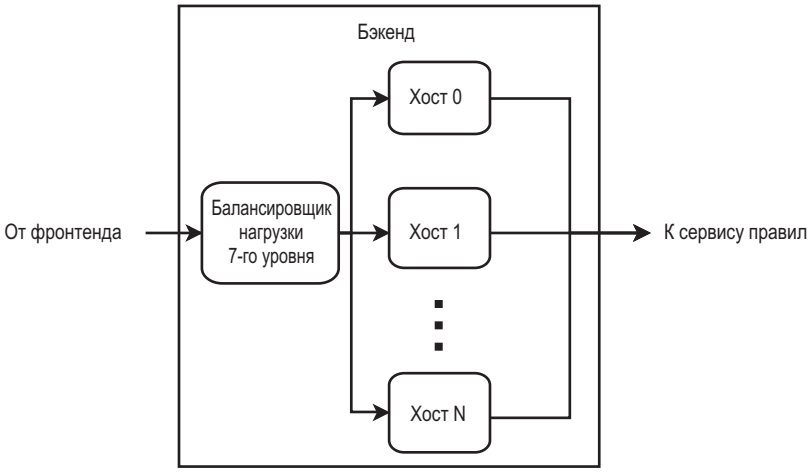

Рис. 8.5. Архитектура бэкенда ограничителя частоты, использующая шардирование с сохранением состояния. Счетчики хранятся в памяти хоста (а не в распределенном кэше, таком как Redis)

При таком подходе сразу возникает вопрос отказоустойчивости, то есть необходимости защищаться от потери данных при отказе хоста. Если такая необходимость существует, она влечет за собой вопросы репликации, обработки отказов, горячих шардов и перераспределения. Как кратко упоминалось в разделе 3.1, можно использовать липкие сеансы в репликации. Но когда выше мы обсуждали требования, мы отмечали, что нам не обязательно нужны консистентность, высокая доступность или отказоустойчивость. Если хост с данными некоторых пользователей выйдет из строя, можно просто закрепить другой хост за этими пользователями и обнулить счетчики запросов. Для нас более важны такие темы, как обнаружение отказа хостов, назначение и предоставление хостов-заменителей и перераспределение трафика.

Ошибка 500 должна инициировать автоматизированное предоставление нового хоста. Новый хост должен получить список адресов из сервиса конфигурации. Это можно сделать либо с помощью простого вручную обновляемого файла, который хранится в распределенном хранилище объектов (именно распределенном, а не на одном хосте, чтобы обеспечить доступность), например AWS S3; либо с помощью более сложного механизма, такого как ZooKeeper. При разработке сервиса ограничения частоты необходимо обеспечить, чтобы настройка хоста не занимала более нескольких минут. Также следует организовать мониторинг на время развертывания хоста и отправлять оповещения с низкой критичностью, если продолжительность настройки превысит несколько минут.

Следует организовать мониторинг горячих шардов и периодически перераспределять трафик между хостами. Можно время от времени запускать пакетное задание ETL, которое читает журналы запросов, находит хосты с большим количеством запросов, определяет подходящую конфигурацию балансировки нагрузки, а затем записывает ее в сервис конфигурации. Задание ETL также может отправить новую конфигурацию в сервис балансировки нагрузки. Мы выполняем запись в сервис конфигурации на случай, если какой-либо из хостов балансировщика нагрузки выйдет из строя. Когда хост восстановится или будет развернут новый хост балансировщика нагрузки, он считает конфигурацию из сервиса конфигурации.

На рис. 8.6 показана архитектура бэкенда с заданием перераспределения. Задание перераспределения предотвращает закрепление большого количества интенсивных пользователей за конкретным хостом, что может привести к выходу его из строя. Так как в нашем решении нет механизмов отработки отказов, которые распределяют пользователей сбойного хоста по другим хостам, отсутствует риск «сваливания в штопор», когда хост выходит из строя из-за избыточного трафика,

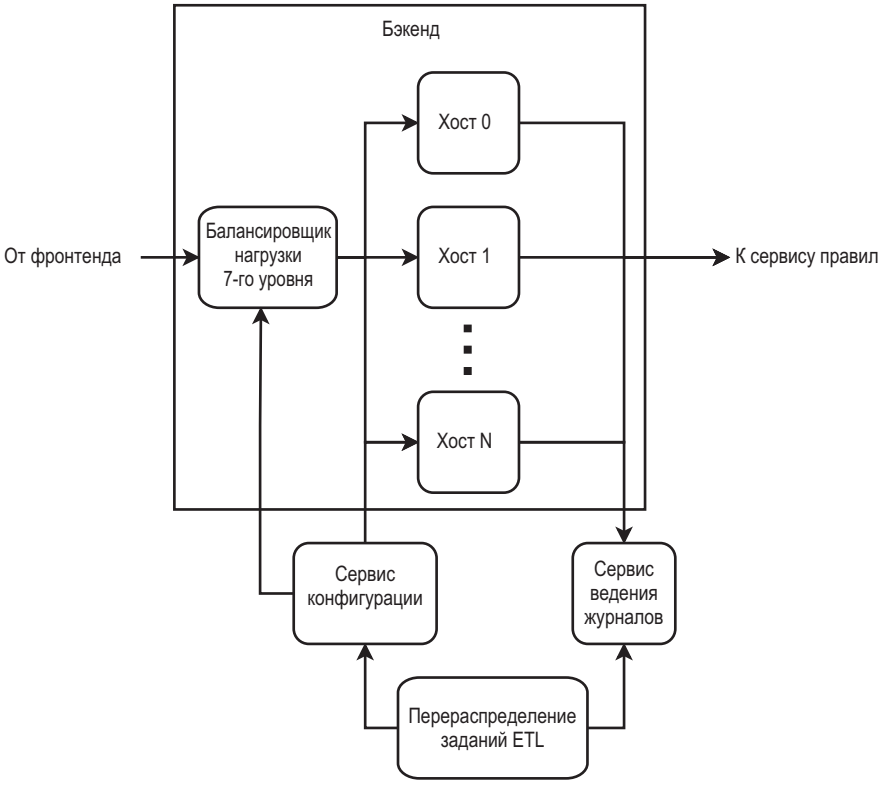

Рис. 8.6. Архитектура бэкенда с ETL-заданием перераспределения трафика

этот трафик перераспределяется между остальными хостами и увеличивает их трафик, что, в свою очередь, приводит к их отказу.

Недостатком такого подхода становится его меньшая гибкость при атаках DoS/DDoS. Если пользователь имеет очень высокую частоту запросов (допустим, сотни запросов в секунду), назначенный хост не справляется с нагрузкой и все пользователи, которым был назначен этот хост, не будут ограничиваться по частоте. Возможно, на такой случай стоит предусмотреть выдачу оповещения и блокировать запросы от этого пользователя по всем сервисам. Балансировщики нагрузки должны блокировать запросы от этого IP-адреса, то есть не отправлять запросы никаким хостам бэкенда и не возвращать никаких ответов, но сохранять в журнале информацию о запросе.

Решение с сохранением состояния получается более сложным, чем решение без сохранения состояния, и обеспечивает более высокую согласованность и точность, но имеет более низкие:

- затраты;
- доступность;
- отказоустойчивость.

В целом этот подход похож на «бедного родственника» распределенной базы данных. Фактически мы проектируем собственное распределенное хранилище, но не такое сложное или зрелое, как популярные распределенные базы данных. Оно оптимизировано для простоты и сохранения низких затрат и не обеспечивает ни сильной согласованности, ни высокой доступности.

8.8. ХРАНЕНИЕ ВСЕХ СЧЕТЧИКОВ НА КАЖДОМ ХОСТЕ

Дизайн бэкенда без сохранения состояния, рассмотренный в разделе 8.6, использует Redis для хранения меток времени запросов. Redis — распределенная база данных с высокой масштабируемостью и доступностью. Кроме того, Redis имеет низкую задержку, и решение будет обладать хорошей точностью. Однако оно потребует использования базы данных Redis, которая обычно реализуется как общий сервис. Нельзя ли избежать зависимости от внешнего сервиса Redis, которая создает риск того, что ограничитель частоты вызовет деградацию этого сервиса?

В варианте с сохранением состояния, рассмотренном в разделе 8.7, этот шаг не требуется благодаря хранению состояния на бэкенде, но в таком случае балансировщик нагрузки должен обрабатывать каждый запрос для определения хоста, которому он должен быть отправлен, а также перетасовки (shuffling), что предотвращает появление горячих шардов. Нельзя ли сократить требования

к хранилищу, чтобы все метки времени пользовательских запросов поместились в памяти на одном хосте?

8.8.1. Высокоуровневая архитектура

Как сократить требования к хранилищу? Чтобы сократить требования с 808 Гбайт до 8,08 Гбайт, или приблизительно 8 Гбайт, можно создать экземпляр сервиса ограничения частоты запросов для каждого из примерно 100 использующих его сервисов и маршрутизировать запросы к соответствующим сервисам через фронтенд. 8 Гбайт данных могут поместиться в памяти хоста. Из-за высокой частоты запросов использовать один хост для ограничения частоты не удастся. Если использовать 128 хостов, на каждом хосте будут храниться всего 64 Мбайта. Скорее всего, в итоге получится от 1 до 128 хостов.

На рис. 8.7 изображена архитектура бэкенда при применении такого подхода. Параллельно с получением запроса хост выполняет следующие действия:

- принимает решение об ограничении частоты и возвращает его;
- синхронизирует в асинхронном режиме с другими хостами свои метки времени.

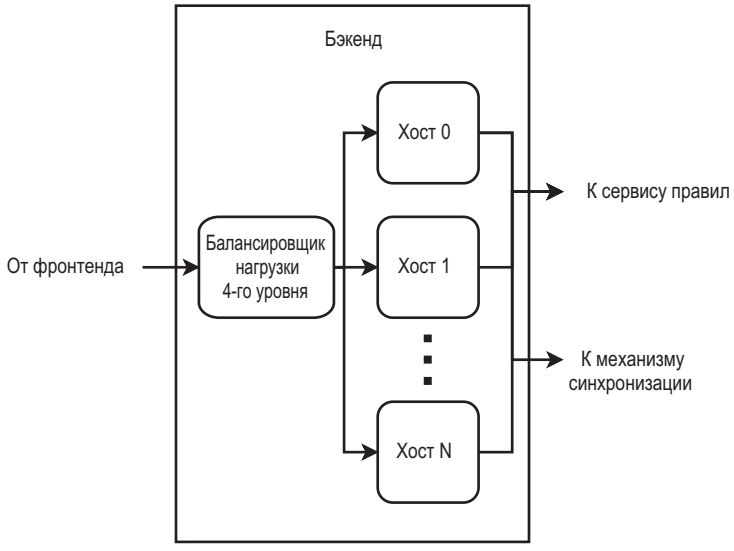

Рис. 8.7. Высокоуровневая архитектура сервиса ограничения частоты, при которой метки времени всех пользовательских запросов находятся на одном хосте бэкенда, запросы случайным образом распределяются между хостами, а каждый хост может выдать в ответ на запрос пользователя решение об ограничении частоты без предварительной отправки запросов к другим сервисам или хостам. Хосты синхронизируют свои метки времени друг с другом в отдельном процессе

Балансировщик нагрузки 4-го уровня случайным образом распределяет запросы между хостами, так что пользователь может направляться на разные хосты при каждом запросе ограничения частоты. Чтобы точно вычислять пороги частот, необходимо поддерживать синхронизацию ограничений частот на хостах. Существует много способов синхронизации хостов, и мы достаточно подробно рассмотрим их в следующем разделе. А пока просто скажем, что будем использовать потоковые обновления вместо пакетных, поскольку пакетные обновления слишком редки и приводят к ограничению пользователей по частоте запросов, значительно превышающей заданную.

По сравнению с двумя другими вариантами дизайна, которые мы обсуждали (дизайн бэкенда без сохранения состояния и дизайн бэкенда с сохранением состояния) этот вариант жертвует согласованностью и точностью ради снижения задержки и повышения производительности (становится возможной обработка более высокой частоты запросов). Так как перед принятием решения об ограничении частоты не все метки времени могут находиться в памяти хоста, вычисленная частота запросов может оказаться ниже фактической. Вот еще особенности этого варианта дизайна:

- Балансировщик нагрузки 4-го уровня используется для направления запросов любому хосту, как с сервисом без сохранения состояния (через фронтенд).
- Хост может принимать решения об ограничении частоты на основании данных, находящихся в памяти.
- Синхронизация данных может выполняться в независимом процессе.

Что, если хост выйдет из строя и его данные будут потеряны? Некоторые пользователи, ограничиваемые по частоте, смогут выдать больше запросов, прежде чем вступит в силу ограничение. Как уже говорилось, это приемлемо. Отработка отказов лидера и возможные проблемы кратко рассматриваются в книге Мартина Клеппмана «Designing Data-Intensive Systems». В табл. 8.2 приведено сравнение трех рассмотренных методов.

Таблица 8.2. Сравнение дизайна бэкенда без сохранения состояния, дизайна бэкенда с сохранением состояния и дизайна с хранением счетчиков на каждом хосте

Дизайн бэкенда без сохранения состояния	Дизайн бэкенда с сохранением состояния	Хранение счетчиков в каждом хосте
Хранит счетчики в распределенной базе данных	Хранит счетчики пользователей в хосте бэкенда	Хранит счетчики пользователей на каждом хосте
Не имеет состояния, так что пользователь может быть направлен на любой хост	Требуется балансировщик нагрузки 7-го уровня для маршрутизации каждого пользователя к назначенному ему хосту	На каждом хосте хранятся счетчики всех пользователей, так что пользователь может быть направлен на любой хост

Дизайн бэкенда без сохранения состояния	Дизайн бэкенда с сохранением состояния	Хранение счетчиков в каждом хосте
Масштабируется. Распределенная база данных используется для обслуживания как высокого трафика чтения, так и высокого трафика записи	Масштабируется. Балансировщик нагрузки — дорогостоящий и вертикально масштабируемый компонент, способный обработать высокую частоту запросов	Не масштабируется, потому что на каждом хосте должны храниться счетчики всех пользователей. Необходимо, чтобы пользователи были разделены между разными экземплярами сервиса и другой компонент (например, фронтенд) направлял пользователей к выделенным им экземплярам
Эффективное потребление хранилища данных. Можно настроить нужный коэффициент репликации в распределенной базе данных	Наименьшее потребление хранилища данных, так как по умолчанию резервное копирование не выполняется. Можно спроектировать сервис хранения с внутрикластерным или внекластерным методом, как обсуждается в разделе 13.5. Без резервного копирования это самый экономичный вариант	Самое дорогостоящее решение. Потребляет большой объем хранилища данных. Также высокий сетевой трафик из-за коммуникаций n-n между хостами для синхронизации счетчиков
Согласованность в конечном счете. Хост, принимающий решение об ограничении частоты, может сделать это до завершения синхронизации, так что решение может быть слегка неточным	Наибольшая точность и согласованность, так как пользователь всегда отправляет запросы к одним и тем же хостам	Наименьшая точность и согласованность, так как синхронизация счетчиков между всеми хостами требует времени
Бэкенд не имеет состояния, поэтому мы используем высокую доступность и отказоустойчивость распределенной базы данных	Без резервного копирования сбой любого хоста приводит к потере данных всех хранящихся на нем счетчиков. Этот вариант дизайна обладает самой низкой доступностью и отказоустойчивостью из трех. Однако эти требования могут быть несущественными, так как относятся к категории нефункциональных. Если ограничитель частоты не может получить точное значение счетчика, он просто пропускает запрос	Хосты взаимозаменяемы, поэтому этот дизайн обладает наибольшей доступностью и отказоустойчивостью из всех трех

Таблица 8.2 (*окончание*)

Дизайн бэкенда без сохранения состояния	Дизайн бэкенда с сохранением состояния	Хранение счетчиков в каждом хосте
Зависит от сервиса внешней базы данных. Неработоспособность таких сервисов может повлиять на ваш сервис, а исправление таких отказов может оказаться неподконтрольным	Не зависит от внешних сервисов баз данных. Балансировщик нагрузки должен обработать каждый запрос, чтобы определить, какому хосту его отправить. Также требует перетасовки для предотвращения появления горячих шардов	Зависит от внешних сервисов баз данных (например, Redis). Отсутствует риск неработоспособности сервиса из-за неработоспособности нижележащих сервисов. Кроме того, проще реализуется, особенно в больших организациях, в которых развертывание или изменение сервисов баз данных сложно проводить из-за бюрократических процедур

8.8.2. Синхронизация счетчиков

Как хостам синхронизировать счетчики пользовательских запросов? В этом разделе мы рассмотрим некоторые возможные алгоритмы. Для нашего ограничителя частоты подойдут любые, кроме алгоритма «все со всеми».

Какую модель синхронизации использовать — pull или push? Можно пожертвовать консистентностью и точностью ради повышения производительности, меньшего потребления ресурсов и снижения сложности. Если хост выйдет из строя, можно просто игнорировать его счетчики и разрешить пользователям выдать больше запросов, пока они не подпадут под ограничение. Учитывая это обстоятельство, можно реализовать асинхронный обмен метками времени через UDP вместо TCP.

Также следует учитывать, что хосты должны обрабатывать трафик от двух основных видов запросов:

1. Запрос на принятие решения об ограничении частоты. Такие запросы ограничиваются балансировщиком нагрузки и выделением большего кластера хостов при необходимости.
2. Запрос на обновление меток времени хоста в памяти. Механизм синхронизации должен следить за тем, чтобы хост не получал слишком частых запросов, особенно при увеличении количества хостов в кластере.

Все со всеми

Принцип *«все со всеми»* означает, что каждый узел передает сообщения всем остальным узлам группы. Это более обобщенный механизм по сравнению

с *широковещательной рассылкой*, которая подразумевает одновременную передачу сообщения многим получателям. На рис. 3.3 (повторенном на рис. 8.8) передача «все со всеми» требует топологии *полноячеистой сети*, при которой каждый узел сети соединен со всеми остальными узлами. Передача «все со всеми» находится в квадратичной зависимости от количества узлов, так что она немасштабируема. Если вы захотите использовать коммуникации «все со всеми» при 128 узлах, это потребует 128 × 128 × 64 Мбайт, то есть > 1 Тбайт памяти, что нереально.

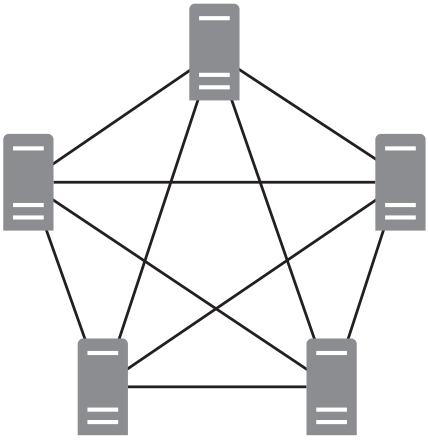

Рис. 8.8. Топология полноячеистой сети. Каждый хост соединен со всеми остальными хостами. В сервисе ограничителя частоты каждый узел получает пользовательские запросы, вычисляет частоту запросов и принимает или отклоняет запрос

Gossip-протокол

В gossip-протоколе, представленном на рис. 3.6 (повтор на рис. 8.9), узлы периодически случайным образом выбирают другие узлы и отправляют им сообщения. Распределенный ограничитель частоты Yahoo использует gossip-протокол для синхронизации своих хостов (https://yahooeng.tumblr.com/post/111288877956/cloud-bouncer-distributed-rate-limiting-at-yahoo). Такой подход жертвует консистентностью и точностью ради повышения производительности и меньшего потребления ресурсов. Кроме того, он повышает сложность.

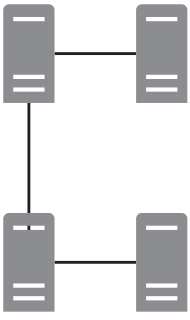

Рис. 8.9. Gossip-протокол. Каждый узел периодически случайным образом выбирает другие узлы и отправляет им сообщения

Схема «все со всеми» и gossip-протокол, рассмотренные в этом разделе, — механизмы синхронизации, в которых все узлы отправляют сообщения напрямую друг другу. Это означает, что все узлы должны знать IP-адреса всех остальных узлов. Так как узлы постоянно добавляются и удаляются из кластера, каждый узел должен выдавать запросы к сервису конфигурации (например, ZooKeeper) для нахождения IP-адресов других узлов.

В других механизмах синхронизации хосты отправляют запросы друг другу через конкретный хост или сервис.

Внешнее хранилище или сервис координации

На рис. 8.10 (который почти не отличается от рис. 3.4) эти два подхода используют внешние компоненты для взаимодействия хостов друг с другом.

Взаимодействие хостов может осуществляться через хост-лидер. Этот хост выбирается сервисом конфигурации кластера (например, ZooKeeper). Каждому хосту достаточно знать IP-адрес хоста-лидера, тогда как хост-лидер должен периодически обновлять свой список хостов.

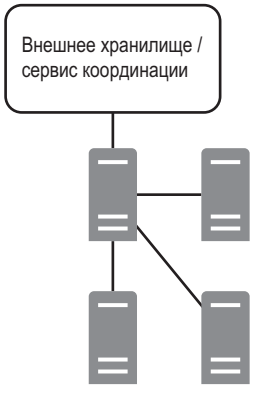

Рис. 8.10. Хосты могут взаимодействовать через внешний компонент, например внешний сервис хранилища / или сервис координации

Случайный выбор лидера

Можно допустить потребление большего количества ресурсов в целях снижения сложности, используя простой алгоритм выбора лидера. На рис. 3.7 (повторенном на рис. 8.11) такой подход приводит к выбору нескольких лидеров. Пока каждый лидер взаимодействует со всеми остальными хостами, каждый хост будет обновляться всеми метками времени запроса. Часто это вызывает дополнительные накладные расходы при пересылке сообщений.

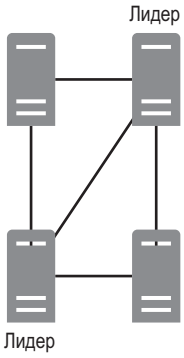

Рис. 8.11. Случайный выбор лидера может привести к выбору нескольких лидеров. Это приведет к повышению расходов ресурсов, но не создаст других проблем

8.9. АЛГОРИТМЫ ОГРАНИЧЕНИЯ ЧАСТОТЫ ЗАПРОСОВ

До сих пор предполагалось, что частота запросов пользователя определялась метками времени запросов, но мы еще не обсуждали возможные методы вычисления частоты запросов. На этом этапе один из главных вопросов — как распределенный сервис ограничения частоты должен определять текущую частоту запросов конкретного источника. Вот некоторые популярные алгоритмы ограничения частоты:

- ведро токенов (token bucket);
- дырявое ведро (leaky bucket);
- счетчик с фиксированным окном (fixed window counter);
- журнал со скользящим окном (sliding window log);
- счетчик со скользящим окном (sliding window counter).

Прежде чем продолжать, заметим, что часть вопросов на собеседованиях о проектировании систем на первый взгляд требуют специальных знаний и опыта, которых нет у большинства кандидатов. Возможно, эксперт не ожидает, что соискатель знаком с алгоритмами ограничения частоты. Он просто хочет оценить навыки коммуникации и способность соискателя к обучению. Эксперт может описать алгоритм ограничения частоты и оценить способность соискателя спроектировать на базе этого алгоритма решение, удовлетворяющее требованиям.

Эксперт даже может делать чрезмерные обобщения или произносить ошибочные утверждения, чтобы оценить вашу способность критически их оценивать и тактично, твердо, ясно и лаконично задавать обоснованные вопросы и выражать свое техническое мнение.

Можно рассмотреть возможность реализации сразу нескольких алгоритмов ограничения частоты, чтобы каждый пользователь сервиса мог выбрать алгоритм, наиболее подходящий для его требований. При таком подходе пользователь выбирает нужный алгоритм и задает конфигурации в сервисе правил.

Для простоты обсуждения в этом разделе будем предполагать, что ограничение частоты составляет 10 запросов за 10 секунд.

8.9.1. Ведро токенов

Алгоритм, показанный на рис. 8.12, основан на аналогии с ведром, заполненным токенами. Ведро обладает тремя характеристиками:

- максимально возможное количество токенов;
- количество токенов, доступных в настоящее время;
- частота заполнения, с которой токены добавляются в ведро.

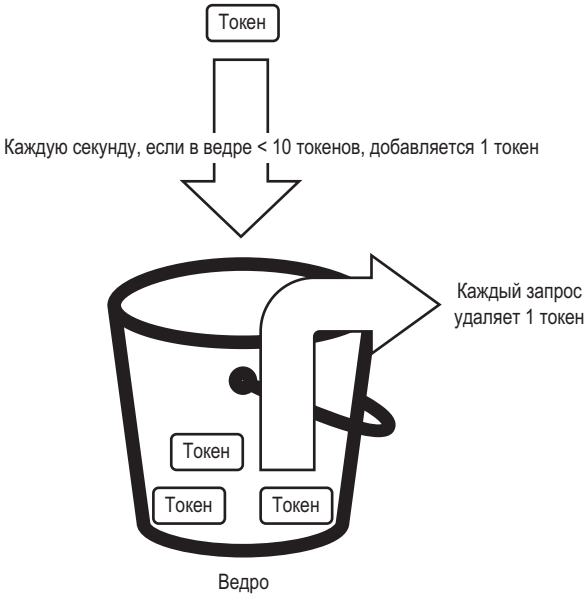

Рис. 8.12. Ведро токенов с ежесекундным пополнением

Каждый раз, когда поступает запрос, из ведра удаляется токен. Если токенов не остается, запрос отклоняется или ограничивается по частоте. Ведро заполняется с постоянной скоростью.

В простой реализации этого алгоритма при отправке каждого пользовательского запроса выполняются следующие действия. Хост хранит пары «ключ — значение»

в хеш-таблице. Если на хосте нет ключа для данного идентификатора пользователя, система инициализирует элемент с идентификатором пользователя и счетчиком токенов 9 (10 – 1). Если на хосте уже существует ключ для идентификатора пользователя и его значение больше 0, система уменьшает его счетчик. Если счетчик достигает 0, возвращается `true`, то есть пользователь должен быть ограничен по частоте. Если возвращается `false`, пользователь ограничиваться не должен. Система также ежесекундно увеличивает каждое значение на 1, если оно меньше 10.

Преимущества алгоритма ведра токенов — его понятность и простота реализации, а также эффективность использования памяти (каждому пользователю нужна всего одна целочисленная переменная для подсчета токенов). Очевидно, каждый хост должен увеличивать каждый ключ в хеш-таблице. Если хеш-таблица находится в памяти хоста, это делается достаточно легко. Если хранилище является внешним по отношению к хосту (как, например, база данных Redis), Redis предоставляет команду MSET (https://redis.io/commands/mset/) для обновления нескольких ключей, однако количество ключей, которые могут быть обновлены одной операцией MSET, может быть ограниченно (https://stackoverflow.com/questions/49361876/mset-over-400-000-map-entries-in-redis). (Stack Overflow не является академически признанным источником, и в официальной документации Redis по MSET не указан верхний лимит количества ключей в запросе. Однако при проектировании системы всегда необходимо задавать обоснованные вопросы, не доверяя полностью даже официальной документации.) Более того, если каждый ключ занимает 64 бита, запрос на обновление 10 миллионов ключей будет иметь размер 8,08 Гбайта, а это слишком много.

Если команду обновления необходимо разделить на несколько запросов, каждый запрос вызывает дополнительные расходы ресурсов и сетевую задержку.

Более того, не существует механизма удаления ключей (то есть удаления пользователей, которые не выдавали запросы в последнее время). А значит, система не знает, когда удалять пользователей для сокращения частоты заполнения токенов или освободить место в базе данных Redis для других пользователей, которые недавно отправляли запросы. Системе понадобится отдельный механизм хранения данных для сохранения последней метки времени запроса пользователя, а также процесс удаления старых ключей.

В распределенной реализации (как в разделе 8.8) каждый хост может содержать собственное ведро токенов и использовать его для принятия решений об ограничении частоты. Хосты могут синхронизировать свои ведра токенов с использованием методов, описанных в разделе 8.8.2. Если хост принимает решение об ограничении частоты, используя свое ведро до его синхронизации с другими хостами, пользователь сможет отправлять запросы с частотой, превышающей заданный лимит. Например, если два хоста получают запросы практически одновременно, на каждом из них после вычитания остаются 9 токенов, после

чего происходит синхронизация с другими хостами. И хотя было два запроса, все хосты синхронизируются до 9 токенов.

Cloud Bouncer

Система Cloud Bouncer (https://yahooeng.tumblr.com/post/111288877956/cloud-bouncer-distributed-rate-limiting-at-yahoo), разработанная в Yahoo в 2014 году, — библиотека распределенного ограничения частоты запросов, основанная на принципе ведра токенов.

8.9.2. Дырявое ведро

В схеме «дырявого ведра» определяется максимальное количество токенов, которые «вытекают» с фиксированной скоростью; когда ведро пустеет, вытекание прекращается. Каждый раз, когда поступает запрос, в ведро добавляется токен. Если ведро заполнено, запрос отклоняется или ограничивается по частоте.

На рис. 8.13 типичная реализация дырявого ведра использует очередь FIFO с фиксированным размером. Из очереди периодически выводятся элементы. При поступлении запроса токен помещается в очередь, если в ней есть свободное место. Из-за фиксированного размера очереди реализация получается менее эффективной по затратам памяти, чем ведро токенов.

Рис. 8.13. Дырявое ведро с выводом из очереди 1 раз в секунду

Алгоритму присущи некоторые недостатки ведра токенов:

- Каждую секунду хост должен выводить элемент из каждой очереди для каждого ключа.
- Необходим отдельный механизм удаления старых ключей.
- Очередь не может превысить свою максимальную емкость, так что в распределенной реализации могут существовать несколько хостов, которые одновременно заполняют свои очереди перед синхронизацией. Это приведет к превышению порога частоты для пользователей.

Другой возможный дизайн основан на метках времени вместо токенов. При поступлении запроса мы сначала выводим метки времени из очереди, пока остальные метки в очереди не станут старше периода удержания, а затем ставим в очередь метку времени запроса, если в очереди остается место. Он возвращает `false`, если постановка в очередь прошла успешно, или `true` в противном случае.

Этот подход позволяет обойтись без обязательного ежесекундного выведения элемента из каждой очереди.

ВОПРОС А вы не заметили потенциальных проблем с согласованностью данных при таком подходе?

Внимательный читатель сразу заметит две возможные проблемы с согласованностью, которые добавляют неточность в решение об ограничении частоты:

1. Может возникнуть ситуация гонки, когда хост записывает пару «ключ — значение» на хост-лидер, и она тут же перезаписывается другим хостом.
2. Часы хостов не синхронизированы, и хост может принять решение об ограничении частоты на основании меток времени других хостов с отличающимися значениями часов.

Небольшая неточность допустима. Эти две проблемы свойственны всем распределенным алгоритмам ограничения частоты, упомянутым в этом разделе, которые используют метки времени (а именно счетчику с фиксированным окном и журналу со скользящим окном), но мы не будем к ним возвращаться.

8.9.3. Счетчик с фиксированным окном

Счетчики с фиксированным окном реализуются в виде пар «ключ — значение». Ключом может быть комбинация идентификатора клиента и метки времени (например, `user0_1628825241`), а значением — счетчик запросов. Когда клиент отправляет запрос, его ключ увеличивается, если он существует, или создается, если не существует. Запрос принимается, если счетчик находится в пределах заданного порога частоты, или отклоняется, если счетчик превышает пороговое значение.

Интервалы окон фиксированы. Например, окно может занимать диапазон [0, 60) секунд каждой минуты. По истечении окна все ключи становятся недействительными. Например, ключ `user0_1628825241` действителен с 3:27:00 GMT до 3:27:59 GMT, потому что 1628825241 соответствует времени 3:27:21 GMT, что лежит в пределах минуты от 3:27 GMT.

ВОПРОС Насколько частота запросов может превысить заданный порог?

Недостаток счетчика с фиксированным окном заключается в том, что частота запросов в нем может вдвое превысить заданный порог. Например, если на рис. 8.13 порог частоты составляет 5 запросов за 1 минуту, клиент может отправить до 5 запросов в период [8:00:00, 8:01:00) и еще до 5 запросов в период [8:01:00, 8:01:30). Получается, что клиент отправил 10 запросов за 1-минутный интервал, — вдвое выше пороговой частоты 5 запросов в минуту (рис. 8.14).

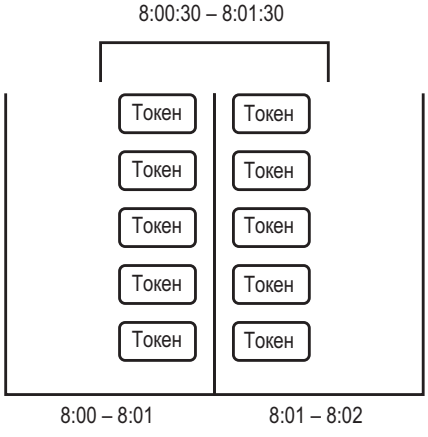

Рис. 8.14. Пользователь отправил 5 запросов в период [8:00:30, 8:01:30) и еще 5 запросов в период [8:01:00, 8:01:30). Получается, что при пороге в 5 запросов в фиксированном окне фактически за 1 минуту было отправлено 10 запросов

В адаптации этого метода для нашего ограничителя частоты каждый раз, когда хост получает пользовательский запрос, он выполняет ряд действий с хеш-таблицей. На рис. 8.15 изображена диаграмма последовательности этих действий:

1. Определение ключей для запроса. Например, если для порога частоты установлен 10-секундный период истечения, соответствующими ключами для `user0` по адресу 1628825250 будут [«user0_1628825241", "user0_1628825242", ..., "user0_1628825250"].
2. Отправка запросов по этим ключам. Если мы храним пары «ключ — значение» в Redis, а не в памяти хоста, можно воспользоваться командой MGET (https://redis.io/commands/mget/) для получения значений всех заданных ключей. И хотя команда MGET имеет сложность O(N), где N — количество получаемых ключей, отправка одного запроса вместо нескольких снижает сетевую задержку и расходы ресурсов.
3. Если ключи не найдены, создается новая пара «ключ — значение», например (`user0_1628825250`, 1). Если найден один ключ, его значение увеличивается. Если найдено более одного ключа (из-за ситуации гонки), значения всех найденных ключей складываются и сумма увеличивается на 1. Результат определяет количество запросов за последние 10 секунд.
4. Параллельно:
   a) Новая или обновленная пара «ключ — значение» записывается на хост-лидер (или в базу данных Redis). Если будет найдено несколько ключей, то удаляются все ключи, кроме самого старого.
   b) Если счетчик превышает 10, то возвращается `true`; в противном случае возвращается `false`.

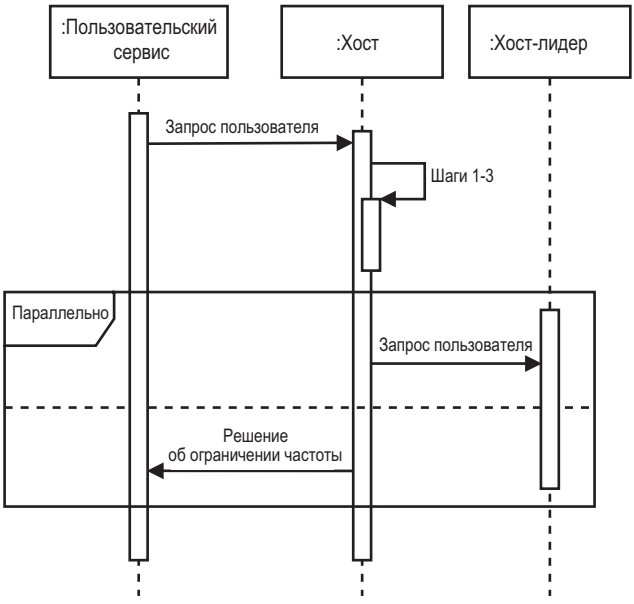

Рис. 8.15. Диаграмма последовательности действий решения со счетчиком с фиксированным окном. Показан вариант с хранением меток времени запросов в памяти хоста, а не в Redis. Решение об ограничении частоты принимается непосредственно на хосте, на основании только данных, хранящихся в памяти хоста. Дальнейшие действия хоста-лидера по синхронизации на диаграмме не представлены

ВОПРОС Как ситуация гонки может привести к обнаружению нескольких ключей на шаге 5?

Для ключей Redis можно определить срок действия (https://redis.io/commands/expire/); настройте ключи так, чтобы они переставали действовать через 10 секунд. В противном случае придется реализовать отдельный процесс для непрерывного поиска и удаления ключей с истекшим сроком. Если такой процесс необходим, то преимущество счетчика с фиксированным окном заключается в том, что процесс удаления ключа не зависит от хостов. Независимый процесс удаления может масштабироваться отдельно от хоста и разрабатываться независимо, что упрощает тестирование и отладку.

8.9.4. Журнал со скользящим окном

Журнал со скользящим окном реализуется парой «ключ — значение» для каждого клиента. Ключом является идентификатор клиента, а значением — отсортированный список меток времени. В журнале со скользящим окном хранятся метки времени всех запросов.

На рис. 8.16 показан простой журнал со скользящим окном. При поступлении нового запроса мы добавляем его метку времени и проверяем, не истек ли срок действия первой метки. Если он истек, выполняется бинарный поиск для нахождения последней истекшей метки, после чего удаляются все предшествующие ей метки. Используйте список вместо очереди, потому что очередь не поддерживает

бинарный поиск. Если список содержит более 10 меток времени, возвращается `true`; в остальных случаях возвращается `false`.

Рис. 8.16. Простой журнал со скользящим окном. Метка времени добавляется при выдаче нового запроса. Затем методом бинарного поиска находится последняя истекшая метка, после чего удаляются все истекшие метки. Запрос разрешен, если размер списка не превышает порогового значения

Журнал со скользящим окном обладает точностью (кроме распределенной реализации из-за факторов, рассмотренных в последнем абзаце раздела 8.9.2), но хранение значения метки времени для каждого запроса потребляет больше памяти, чем ведро токенов.

Алгоритм журнала со скользящим окном подсчитывает запросы даже после превышения порога частоты, так что с его помощью можно измерить фактическую частоту запросов пользователя.

8.9.5. Счетчик со скользящим окном

Счетчик со скользящим окном — следующая стадия развития алгоритма счетчика с фиксированным окном и журналом со скользящим окном. В нем используются несколько интервалов фиксированных окон, и каждый интервал составляет 1/60 длины временного окна порога частоты.

Например, если период ограничения частоты составляет 1 час, мы используем 60 окон продолжительностью 1 минуту вместо одного 1-часового окна. Текущая частота определяется сложением последних 60 окон. Такой способ может привести к небольшому занижению числа запросов. Например, подсчет запросов с 11:00:35 суммирует 60 минутных окон от [10:01:00, 10:01:59] до [11:00:00, 11:00:59] и проигнорирует окно [10:00:00, 10:00:59]. Впрочем, этот способ все равно точнее счетчика с фиксированным окном.

8.10. ПРИМЕНЕНИЕ ПАТТЕРНА SIDECAR

Мы пришли к возможности применения паттерна sidecar для политик ограничения частоты. Сервис ограничения частоты с применением паттерна sidecar показан на рис. 1.8. По аналогии с тем, что говорилось в разделе 1.4.6, администратор может настроить политики ограничения частоты пользовательского сервиса в области управления, так чтобы распределить их по sidecar-хостам. В таком дизайне, где пользовательские сервисы хранят политики ограничения частоты

на своих sidecar-хостах, хостам пользовательских сервисов не нужно отправлять запросы к сервису ограничения частоты для получения политик ограничения, что избавляет от расходов сетевых ресурсов, связанных с такими запросами.

8.11. ВЕДЕНИЕ ЖУРНАЛОВ, МОНИТОРИНГ И ОПОВЕЩЕНИЯ

Кроме практик ведения журналов, мониторинга и оповещений, рассмотренных в разделе 2.5, следует настроить мониторинг и оповещения для перечисленных ниже событий. Можно настроить задачи мониторинга в общем сервисе мониторинга для журналов общего сервиса ведения журналов, и эти задачи должны инициировать уведомления в общем сервисе уведомлений, чтобы оповещать разработчиков о следующих проблемах:

- признаки возможной вредоносной активности — например, пользователи, которые продолжают отправлять запросы с высокой частотой, несмотря на скрытую блокировку;
- признаки возможных попыток DDoS — например, аномально высокое количество пользователей, ограничиваемых по частоте за короткий период времени.

8.12. ПРЕДОСТАВЛЕНИЕ ФУНКЦИОНАЛЬНОСТИ В КЛИЕНТСКОЙ БИБЛИОТЕКЕ

Нужно ли пользовательскому сервису обращаться с запросом к сервису ограничения частоты для каждого запроса? Или агрегировать пользовательские запросы и обращаться к сервису ограничения частоты только при наступлении определенных условий, например:

- накоплении пакета пользовательских запросов;
- внезапного повышения частоты запросов?

Обобщая сказанное, можно ли реализовать ограничение частоты в форме библиотеки, а не сервиса? В разделе 6.7 приведено общее сравнение библиотек с сервисами. Если полностью реализовать эту функциональность в форме библиотеки, следует использовать подход из раздела 8.7, когда хост может хранить все запросы пользователя в памяти и синхронизировать их друг с другом. Хосты должны иметь возможность взаимодействовать друг с другом для синхронизации меток времени пользовательских запросов, а следовательно, разработчики сервиса, использующего вашу библиотеку, должны настроить сервис конфигурации, такой как ZooKeeper. Для многих разработчиков это будет слишком сложно и чревато ошибками, поэтому в качестве альтернативы можно предложить библиотеку

с возможностью повышения производительности сервиса и ограничения частоты за счет переноса части обработки на сторону клиента (для сокращения частоты запросов к сервису).

Паттерн распределения обработки между клиентом и сервисом можно обобщить для любой системы, но он может создать сильную связанность между клиентом и сервисом, которая в общем случае считается антипаттерном. Серверное приложение должно поддерживать старые версии клиентов в течение долгого времени. По этой причине клиентский SDK (Software Development Kit) обычно представляет собой лишь слой для набора конечных точек REST или RPC и не выполняет никакой обработки данных. Для выполнения обработки данных на стороне клиента должно соблюдаться по крайней мере одно из следующих условий:

- Обработка должна быть простой, чтобы эту клиентскую библиотеку было легко поддерживать в будущих версиях серверного приложения.
- Обработка связана с высокими затратами ресурсов, так что расходы на обслуживание такой обработки на стороне клиента оправдываются значительным снижением денежных затрат на обслуживание сервиса.
- Должен существовать четко заявленный жизненный цикл поддержки с понятным информированием пользователей о прекращении поддержки клиента.

Что касается пакетирования запросов к ограничителю частоты, можно поэкспериментировать с размером пакета, чтобы найти оптимальное соотношение между точностью и сетевым трафиком.

А если клиент также измеряет частоту запросов и использует сервис ограничения частоты только в случае превышения заданного порога? Недостаток такого решения в том, что поскольку клиенты не взаимодействуют друг с другом, клиент может измерить только частоту запросов к конкретному хосту, на котором он установлен, и не может измерять частоту запросов отдельных пользователей. Это означает, что ограничение частоты срабатывает на основании частоты запросов по всем пользователям, а не для отдельных пользователей. Пользователи привыкают к определенному пределу частоты и начинают жаловаться, если вдруг сталкиваются с ограничениями частоты, которых раньше не было.

Возможно альтернативное решение: обнаружение аномалий на стороне клиента для выявления неожиданного роста частоты запросов и последующая отправка сервису запросов на ограничение частоты.

8.13. ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

- Михаил Смарщок (Mikhail Smarshchok), 2019. YouTube-канал System Design Interview, https://youtu.be/FU4WlwfS3G0.

- Обсуждения алгоритмов счетчика с фиксированным окном, журнала со скользящим окном и счетчика со скользящим окном адаптированы по материалам https://www.igma.com/blog/an-alternative-approach-to-rate-limiting/.
- Нил Мадден (Neil Madden). «API Security in Action». Manning Publications, 2020.
- Кристиан Поста (Christian Posta), Ринор Малоку (Rinor Maloku). «Istio in Action». Manning Publications, 2022.
- Морган Брюс (Morgan Bruce), Паоло Перейра (Paulo Pereira). «Microservices in Action», глава 3.5. Manning Publications, 2018.

ИТОГИ

- Ограничение частоты запросов позволяет избежать отказов сервисов и ненужных затрат.
- Такие альтернативы, как добавление хостов или использование балансировщика нагрузки для ограничения частоты запросов, обычно не удовлетворяют требованиям. Добавление хостов для обработки выбросов трафика слишком медленное, а применение балансировщика нагрузки 7-го уровня только для ограничения частоты запросов может значительно увеличить затраты и сложность.
- Не используйте ограничение частоты запросов, если оно ухудшает пользовательский опыт, а также в сложных сценариях (например, с подписками).
- Нефункциональные требования ограничителей частоты запросов — масштабирование, производительность и низкая сложность. Чтобы оптимизировать систему для этих требований, можно пожертвовать доступностью, отказоустойчивостью, точностью и консистентностью.
- Основные входные данные для сервиса ограничителя частоты запросов — идентификатор пользователя и идентификатор сервиса. Они обрабатываются в соответствии с правилами, определяемыми администраторами, чтобы сервис возвращал ответ («да» или «нет») как решение об ограничении частоты.
- Существуют разные алгоритмы ограничения частоты запросов, у каждого из которых свои достоинства и недостатки. Алгоритм ведра токенов понятен, прост в реализации и экономно расходует память, но имеет недостатки синхронизации и очистки. Алгоритм дырявого ведра понятен и прост в реализации, но ему не хватает точности. Журнал с фиксированным окном отличается простотой тестирования и отладки, но он неточен и более сложен в реализации. Журнал со скользящим окном точен, но требует больше памяти. Счетчик со скользящим окном расходует меньше памяти, но менее точен, чем журнал со скользящим окном.
- Для сервиса ограничения частоты запросов можно рассмотреть паттерн sidecar.

9 Проектирование сервиса уведомлений/оповещений

В ЭТОЙ ГЛАВЕ

- ✓ Ограничение функциональности и обсуждение дизайна сервиса
- ✓ Проектирование сервиса, делегирующего выполнение платформенно-зависимым каналам
- ✓ Проектирование системы для гибких конфигураций и шаблонов
- ✓ Другие стандартные обязанности сервиса

Мы определяем функции и классы в исходном коде, чтобы избежать дублирования при программировании, отладке и тестировании; тем самым мы упрощаем обслуживание и делаем возможным повторное использование кода. Аналогичным образом происходит обобщение функциональности, используемой несколькими сервисами (централизация сквозной функциональности).

Отправка уведомлений (notifications) и оповещений (alerts) пользователям относится к общим системным требованиям. При любом обсуждении дизайна системы, когда речь заходит об отправке уведомлений, следует предложить общий сервис уведомлений для организации.

9.1. ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

Сервис уведомлений должен быть по возможности простым и предназначенным для широкой категории пользователей, что создает значительную сложность в функциональных требованиях. Такой сервис уведомлений может предоставлять целый ряд возможностей. С учетом ограниченного времени следует четко определить основные сценарии использования сервиса и функции, которые сделают его полезным для широкого круга пользователей. Четкое определение функциональности позволит провести оптимизацию нефункциональных требований сервиса. После подготовки дизайна исходной версии системы можно обсудить и спроектировать дополнительные возможные функции.

Это может стать хорошим упражнением на проектирование минимально жизнеспособного продукта (MVP, Minimum Viable Product). Можно спланировать подходящую функциональность и спроектировать систему в виде набора слабосвязанных компонентов, чтобы она могла адаптироваться к добавлению новой функциональности и сервисов и развиваться в ответ на обратную связь пользователей и на изменения бизнес-требований.

9.1.1. Не для мониторинга работоспособности

Скорее всего, сервис уведомлений станет слоем на вершине иерархии разных систем передачи сообщений (электронной почты, SMS и т. д.). Сервис для отправки таких сообщений (например, сервис электронной почты) сам по себе достаточно сложен. В рамках текущей задачи мы будем использовать общие сервисы передачи сообщений, но не будем проектировать их. Мы спроектируем сервис для пользователей, предназначенный для отправки сообщений по разным каналам.

Применение этого подхода за пределами общих сервисов передачи сообщений позволит использовать другие общие сервисы для такой функциональности, как хранение данных, потоковая передача событий и ведение журналов. Кроме того, можно использовать существующую инфраструктуру с общим доступом (физическое оборудование или облачную инфраструктуру), которая применяются в организации для разработки других сервисов.

> **ВОПРОС** Можно ли реализовать мониторинг работоспособности с использованием существующей инфраструктуры с общим доступом или сервисов, которые он отслеживает?

Руководствуясь таким подходом, мы предполагаем, что этот сервис не должен применяться для мониторинга работоспособности (то есть отправлять оповещения при сбоях других сервисов). В противном случае его нельзя создавать на базе существующей инфраструктуры или с использованием общих сервисов организации, потому что их отказы повлекут отказ и этого сервиса и оповещения о неработоспособности отправляться не будут. Сервис мониторинга

работоспособности должен работать на инфраструктуре, не зависящей от отслеживаемых им сервисов. Это одна из ключевых причин популярности внешних сервисов мониторинга работоспособности, таких как PagerDuty.

В разделе 9.14 рассматривается возможное использование сервиса для мониторинга работоспособности.

9.1.2. Пользователи и данные

У нашего сервиса уведомлений три типа пользователей:

- Отправитель: человек или сервис, который выполняет CRUD-операции (create, read, update, delete — создание, чтение, обновление, удаление) с уведомлениями и отправляет уведомления получателям.
- Получатель: пользователь приложения, получающий уведомления. Мы также будем относить к категории получателей устройства или сами приложения.
- Администратор: человек, имеющий административный доступ к сервису уведомлений. Администратор обладает различными привилегиями. Он может предоставлять другим пользователям разрешения на получение или отправку уведомлений, а также создавать шаблоны уведомлений и управлять этими шаблонами (раздел 9.5). Предполагается, что вы как разработчик сервиса уведомлений обладаете административным доступом, хотя на практике лишь часть разработчиков имеет административный доступ к рабочей среде сервиса.

Отправители могут быть как физическими, так и программными. Программные пользователи могут использовать запросы API, особенно для отправки уведомлений. Физические пользователи реализуют все свои сценарии использования, включая отправку уведомлений, а также такие административные функции, как настройка уведомлений и просмотр отправленных и ожидающих уведомлений, через веб-интерфейс.

Размер уведомлений можно ограничить 1 Мбайт, этого более чем достаточно для тысяч символов и небольшого изображения. Пользователи не могут отправлять в уведомлениях ни видео, ни аудио. Вместо этого в уведомление включается ссылка на мультимедийный контент или любые другие большие файлы, а в системе получателя должны присутствовать средства, разработанные отдельно от сервиса уведомлений, для загрузки и просмотра этого контента. Взломщики могут пытаться прислать подложные уведомления со ссылками на вредоносные веб-сайты. Чтобы этого не произошло, уведомление должно содержать цифровую подпись. Получатели могут проверить подпись у поставщика сертификатов. Дополнительную информацию можно найти на ресурсах, посвященных криптографической защите.

9.1.3. Каналы получателей

Сервис должен поддерживать возможность отправки уведомлений по разным каналам, включая перечисленные ниже. Сервис уведомлений должен быть интегрирован с сервисами, отправляющими сообщения по каждому из следующих каналов:

- браузер;
- электронная почта;
- SMS (для простоты мы не рассматриваем MMS);
- автоматизированные телефонные звонки;
- push-уведомления для Android, iOS и браузеров;
- специализированные уведомления в приложениях: например, банковские или финансовые приложения с жесткими требованиями к конфиденциальности и безопасности используют внутренние системы передачи сообщений и уведомлений.

9.1.4. Шаблоны

Каждая отдельная система передачи сообщений предоставляет шаблон по умолчанию с набором полей, которые заполняются пользователем перед отправкой сообщения. Например, в электронной почте присутствует поле адреса отправителя, поле адреса получателя, поле темы, поле тела и список вложений; в SMS присутствует поле телефона отправителя, поле телефонов получателей и поле тела.

Одно уведомление может рассылаться многим получателям. Например, приложение может отправить электронную почту или push-уведомление с приветствием для нового пользователя, только что зарегистрировавшегося в приложении. Сообщение может быть одинаковым для всех пользователей, например: Welcome to Beigel. Please enjoy a 20% discount on your first purchase[1].

Сообщение также может содержать параметры персонализации, например имя пользователя и процент скидки (Welcome ${first_name}. Please enjoy a ${discount}% discount on your first purchase[2]). Другой пример — подтверждение заказа в сообщении электронной почты, тексте или push-уведомлении, отправляемом интернет-магазином клиенту сразу же после оформления заказа. Сообщение также может иметь параметры: имя клиента, код подтверждения заказа, список товаров (товар может иметь несколько параметров) и цены.

---

[1] Добро пожаловать в Бейгл! Получите скидку 20% на первый заказ! — *Примеч. пер.*

[2] Добро пожаловать в Бейгл, ${имя}! Получите скидку ${размер_скидки}% на первый заказ! — *Примеч. пер.*

Сервис уведомления может предоставлять API для шаблонов CRUD. Каждый раз, когда пользователь захочет отправить уведомление, он может либо создать все сообщение самостоятельно, либо выбрать подходящий шаблон и заполнить значения для этого шаблона.

Поддержка шаблонов также сокращает трафик к сервису уведомлений. Эта тема обсуждается ниже в этой главе.

Вы можете предоставить большое количество инструментов для создания и управления шаблонами и выделить их в отдельный сервис (сервис шаблонов). Мы же ограничимся шаблонами CRUD.

9.1.5. Условия инициирования

Уведомления могут инициироваться вручную или на программном уровне. Вы можете предоставить браузерное приложение, в котором пользователь создает уведомления, добавляет получателей, а затем сразу же отправляет уведомление. Уведомления также могут рассылаться на программном уровне; рассылка может происходить по настроенному расписанию или по запросу API.

9.1.6. Управление подписчиками, группы отправителей и группы получателей

Если пользователь может отправить уведомление сразу нескольким получателям, стоит предоставить средства для управления группами получателей. Пользователь может адресовать уведомление группе получателей, чтобы не перечислять всех получателей при каждой отправке.

> **ПРЕДУПРЕЖДЕНИЕ** Группы получателей содержат персональные данные, поэтому на них распространяются законы конфиденциальности (такие, как GDPR и CCPA).

Пользователи должны иметь возможность выполнять операции CRUD с группами получателей. Рассмотрите вариант управления доступом на основе ролей (RBAC, Role-Based Access Control). Например, в группе могут быть определены роли чтения и записи. Пользователь должен иметь роль чтения для просмотра групп и других подробностей, а затем роль записи для добавления или удаления участников. Мы не будем подробно рассматривать организацию RBAC для групп.

Получатель должен иметь возможность согласиться на получение уведомлений или отказаться от них; в противном случае они станут обычным спамом. В данной главе эта тема не рассматривается. На собеседовании о ней может зайти речь в дополнение к основной теме.

9.1.7. Функции пользователя

Другие функции, которые можно реализовать:

- Сервис должен выявлять дубликаты запросов уведомлений от отправителей и не отправлять дубликаты уведомлений получателям.
- Пользователь может просмотреть свои прошлые запросы уведомлений. В одном из важных сценариев использования пользователь проверяет, отправлял ли он уже определенный запрос уведомления, чтобы избежать дубликатов запросов. И хотя сервис уведомлений также может автоматически выявлять дубликаты запросов уведомлений, полностью полагаться на эту реализацию нельзя, так как пользователь может задать запрос-дубликат отдельно от сервиса уведомлений.
- Пользователь может хранить многочисленные конфигурации уведомлений и шаблоны. Ему должны быть доступны средства поиска конфигураций или шаблонов по различным полям, например именам или описаниям. Также у пользователя должна быть возможность сохранения нужных уведомлений.
- Пользователь должен иметь возможность просматривать статусы уведомлений. Уведомление может быть запланировано, находиться в обработке (как с сообщениями в папке исходящей почты) или не доставлено. В случае неудачной доставки уведомления пользователь должен иметь возможность посмотреть, запланирована ли повторная доставка, а также количество повторных попыток доставки.
- (Не обязательно). Уровень приоритета, назначенный пользователем. Например, сервис может обрабатывать высокоприоритетные уведомления перед низкоприоритетными или применять веса для предотвращения зависания.

9.1.8. Аналитика

Мы будем исходить из того, что аналитика выходит за пределы нашей задачи, хотя ее можно обсудить при проектировании сервиса уведомлений.

9.2. НЕФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

Можно обсудить следующие нефункциональные требования:

- Масштабирование: сервис уведомлений должен быть способен отправлять миллиарды уведомлений ежедневно. При 1 Мбайт на уведомление сервис будет ежедневно обрабатывать и отправлять петабайты данных. В системе могут быть тысячи отправителей и миллиарды получателей.
- Производительность: уведомления должны доставляться за секунды. Чтобы улучшить скорость доставки критических уведомлений, можно разрешить

пользователям устанавливать для некоторых сообщений более высокий приоритет.

- Высокая доступность: работоспособность на уровне «пять девяток».
- Отказоустойчивость: если получатель не может получить уведомление, он должен его получить при ближайшей возможности.
- Безопасность: только авторизованные пользователи должны иметь возможность отправлять уведомления.
- Конфиденциальность: получатели должны иметь возможность отказаться от получения уведомлений.

9.3. ИСХОДНАЯ ВЫСОКОУРОВНЕВАЯ АРХИТЕКТУРА

При проектировании системы можно руководствоваться следующими соображениями:

- Пользователь, запросивший создание уведомлений, делает это через единый сервис с единым интерфейсом. Пользователи выбирают нужный канал(-ы) и другие параметры через этот единый сервис/интерфейс.
- Тем не менее каждый канал может обслуживаться отдельным сервисом. Каждый сервис канала предоставляет логику, относящуюся к этому каналу. Например, сервис канала уведомлений через браузер может создавать браузерные уведомления через API веб-уведомлений. Подробности см. в документации: например, Using the Notifications API (https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API/Using_the_Notifications_API) и Notification (https://developer.mozilla.org/en-US/docs/Web/API/notiication). Некоторые браузеры (например, Chrome) также предоставляют собственный API уведомлений. За информацией о расширенных уведомлениях с такими элементами, как графика и индикаторы прогресса, обращайтесь к chrome.notifications (https://developer.chrome.com/docs/extensions/reference/notifications/) и Rich Notifications API (https://developer.chrome.com/docs/extensions/mv3/richNotiications/).
- Общую логику сервиса канала можно централизовать в другом сервисе, называемом «конструктор задач».
- Уведомления по разным каналам могут обслуживаться внешними сторонними сервисами, как показано на рис. 9.1. Для выдачи push-уведомлений в Android используется механизм Firebase Cloud Messaging (FCM), а в iOS — сервис уведомлений Apple Push. Также можно применить сторонние сервисы для электронной почты, SMS, текстовых сообщений и телефонных звонков. Отправка запросов к сторонним сервисам означает, что частоту запросов необходимо ограничить и обрабатывать недоставленные запросы.
- Отправка уведомлений исключительно через синхронные механизмы не масштабируется, потому что процесс потребляет поток во время ожидания

отправки запросов и ответов по сети. Чтобы поддерживать тысячи отправителей и миллиарды получателей, следует применять асинхронные механизмы (такие, как потоковая передача сообщений).

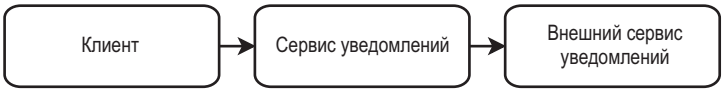

Рис. 9.1. Наш сервис уведомлений может отправлять запросы к внешним сервисам уведомлений, поэтому он должен ограничить частоту запросов и обрабатывать недоставленные запросы

На рис. 9.2 и 9.3 изображена исходная высокоуровневая архитектура. Чтобы отправить уведомление, клиент выдает запрос к сервису уведомлений. Запрос сначала обрабатывается сервисом фронтенда или шлюзом API, после чего отправляется сервису бэкенда. Сервис бэкенда связан с кластером-производителем, топиком уведомлений Kafka и кластером-потребителем. Хост-производитель просто отправляет сообщение в топик уведомлений Kafka и возвращает код успеха 200. Кластер-потребитель потребляет сообщения, генерирует события уведомлений и помещает их в соответствующие очереди каналов. Каждое событие уведомления предназначено для одного получателя/приемника. Асинхронный событийный механизм позволяет сервису уведомлений справляться с непредсказуемыми выбросами трафика.

На другом конце очереди для каждого канала уведомлений существует отдельный сервис. Некоторые из них могут зависеть от внешних сервисов, таких как Firebase Cloud Messaging (FCM) для Android и Apple Push Notiication Service (APNs) для iOS. Браузерный сервис уведомлений может дополнительно делиться по разным типам браузеров (например, Firefox или Chrome).

Каждый канал уведомлений должен быть реализован как отдельный сервис (будем называть его сервисом канала), потому что отправка уведомлений в конкретный канал требует конкретного сервиса приложений, и все каналы обладают разными возможностями, конфигурациями и протоколами. Уведомления по электронной почте используют протокол SMTP. Чтобы отправить уведомление через систему электронной почты, пользователь указывает адрес отправителя, адрес получателя, заголовок, тело и вложения. Также существуют другие типы электронной почты, например календарные события. Шлюз SMS использует различные протоколы, включая HTTP, SMTP и SMPP. Чтобы отправить сообщение SMS, пользователь указывает номер отправителя, номер получателя и строку.

В этом обсуждении термином «приемник» или «адрес» будет обозначаться поле, указывающее, куда следует отправить объект уведомления; это может быть номер телефона, адрес электронной почты, идентификатор устройства для push-уведомлений или нестандартные получатели (например, идентификатор пользователя для внутренней передачи сообщений и т. д.).

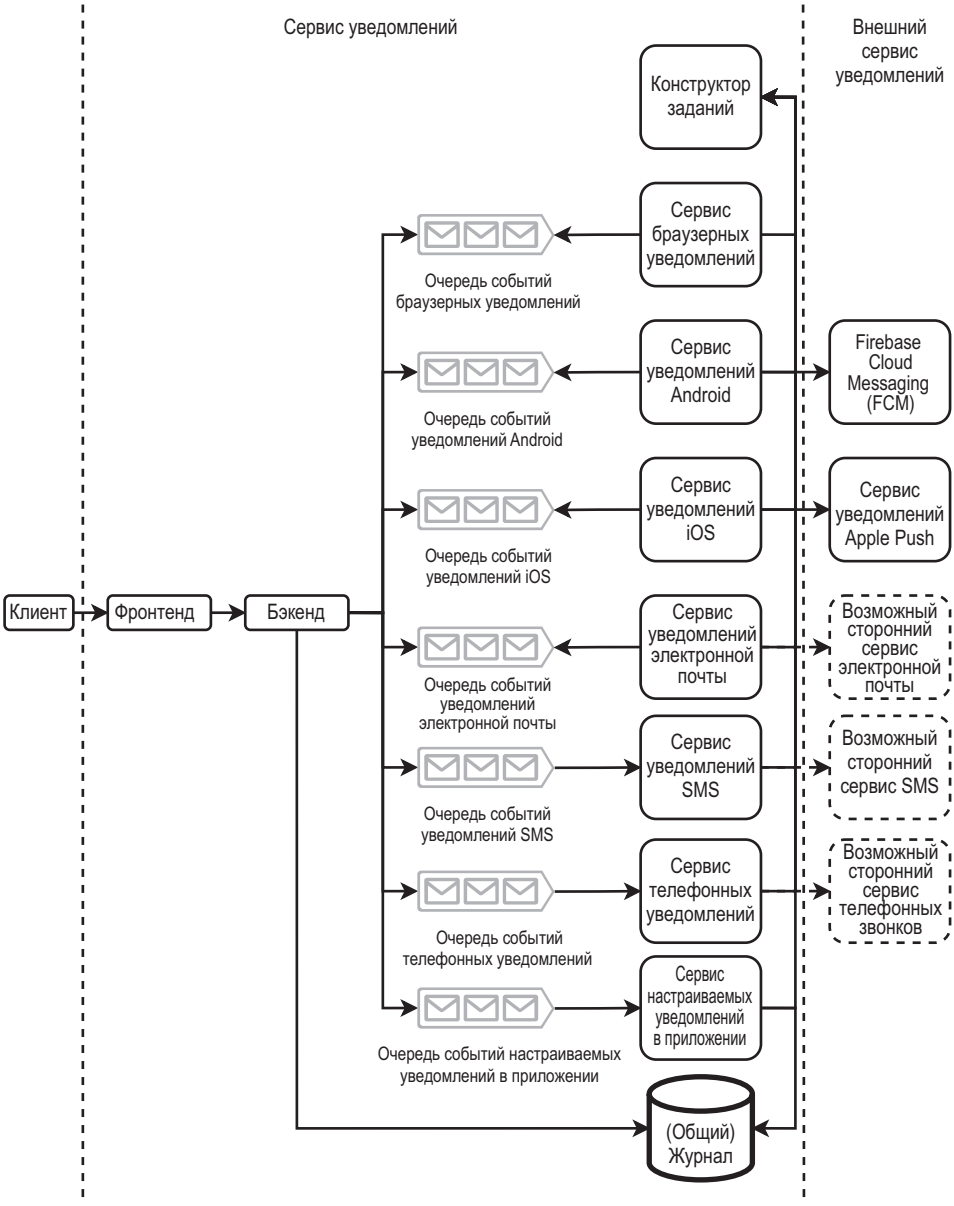

Рис. 9.2. Высокоуровневая архитектура сервиса уведомлений со всеми возможными запросами при отправке уведомлений клиентом/пользователем. Разные потребители Kafka (каждый из которых является сервисом уведомлений для конкретного канала) обозначаются общим термином «сервисы каналов». На диаграмме показано, что сервисы бэкенда и сервисы каналов используют общую базу данных ведения журналов, но все компоненты сервиса уведомлений должны вести журналы через общий сервис журналирования

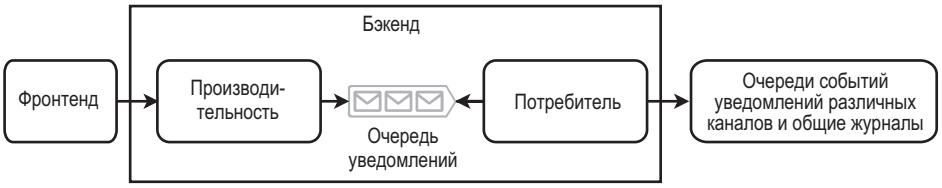

Рис. 9.3. Увеличенное изображение сервиса внутренней части на рис. 9.2. Сервис бэкенда состоит из кластера-производителя, темы уведомлений Kafka и кластера-производителя. На дальнейших диаграммах увеличенная диаграмма сервиса бэкенда не приводится

Каждый сервис канала должен выполнять базовую функциональность отправки уведомлений приемнику. Он должен обрабатывать весь контент уведомлений и доставлять уведомления приемнику. Но возможно, для доставки сообщений по некоторым каналам придется использовать сторонние API. Например, можно воспользоваться API телекоммуникационной компании для проведения телефонных звонков и SMS. Для активных уведомлений на мобильных устройствах используется сервис Apple Push для iOS, и Firebase Cloud Messaging для Android. Только для браузерных уведомлений и уведомлений через специальное приложение возможна доставка сообщений без использования сторонних API. При использовании стороннего API возможность отправлять к нему запросы напрямую должна быть только у соответствующего сервиса канала.

Отсутствие связанности между сервисом канала и другими сервисами сервиса уведомлений улучшает отказоустойчивость системы и открывает следующие возможности:

- Сервис канала может использоваться другими сервисами помимо сервиса уведомлений.
- Сервисы могут изменять детали своей внутренней реализации независимо друг от друга и их могут обслуживать специализированные команды. Например, команда сервиса телефонных звонков должна отвечать за совершение автоматизированных звонков, а группа сервиса электронной почты — за отправку электронной почты, но им не нужно знать, как работает сервис другой команды.
- В системе могут присутствовать специализированные сервисы каналов, и сервис уведомлений должен отправлять им запросы. Например, можно реализовать уведомления в браузере или мобильном приложении, которые отображаются в виде нестандартных UI-компонентов вместо push-уведомлений. Модульная структура сервисов каналов упрощает их разработку.

Вы можете использовать аутентификацию (см. обсуждение OpenID Connect в приложении) с сервисом интерфейсной части, чтобы убедиться в том, что только авторизованные пользователи (такие, как хосты уровня сервиса) могут

обращаться с запросами к сервисам каналов для отправки уведомлений. Сервис фронтенда обрабатывает запросы к сервису авторизации OAuth2.

Но почему пользователи просто не могут использовать системы уведомлений нужных им каналов? Какие преимущества компенсируют лишние затраты на разработку и обслуживание дополнительных уровней?

Сервис уведомлений предоставляет общий пользовательский интерфейс своим клиентам (то есть сервисам каналов), чтобы пользователи могли управлять всеми своими уведомлениями по всем каналам из одного сервиса и им не нужно было изучать несколько сервисов и управлять ими. Сервис фронтенда предоставляет общий набор операций:

- *Ограничение частоты запросов* — предотвращает ошибки 5xx из-за того, что клиенты уведомлений перегружены слишком большим количеством запросов. Ограничение частоты может быть отдельным общим сервисом — его мы рассмотрели в главе 8. Для определения лимита запросов можно воспользоваться нагрузочным тестированием. Ограничитель частоты также может оповещать специалистов по обслуживанию о том, что частота запросов конкретного канала стабильно превышает или находится значительно ниже заданного лимита и следует принять соответствующее решение о масштабировании. Еще одна возможность, которую стоит рассмотреть, — автомасштабирование.
- *Конфиденциальность* — в организациях могут существовать конкретные политики конфиденциальности, определяющие правила отправки уведомлений устройствам или учетным записям. Уровень сервиса может использоваться для настройки и обеспечения соблюдения этих политик всеми клиентами.
- *Безопасность* — аутентификация и авторизация для всех уведомлений.
- *Мониторинг, аналитика и уведомления* — сервис может регистрировать события уведомлений и агрегировать статистику (например, частоты успешных или неудачных попыток доставки уведомлений в скользящих окнах разной ширины). Пользователи могут отслеживать эти статистические показатели и устанавливать пороги оповещений для частот отказов.
- *Кэширование* — запросы могут осуществляться через сервис кэширования. При этом используется одна из стратегий кэширования, описанных в главе 8.

Для каждого канала предоставляется топик Kafka. Если уведомление имеет несколько каналов, можно производить событие для каждого канала и в соответствующий топик. Также можно создать топик Kafka для каждого уровня приоритета, так что при наличии пяти каналов и трех уровней приоритета понадобится 15 тем.

Решение с использованием Kafka вместо синхронной схемы «запрос — ответ» следует нативному облачному принципу событийного управления вместо синхронного. Он имеет целый ряд преимуществ: слабую связанность, независимую разработку разных компонентов в сервисе, упрощение диагностики (можно воспроизвести прошлые сообщения в любой момент времени) и более высокую пропускную способность без блокирующих вызовов. С другой стороны, это повышает затраты на хранение данных. Если вы обрабатываете 1 миллиард сообщений ежедневно, требования к объему хранилища составляют 1 Пбайт за день, или приблизительно 10 Пбайт за недельный период удержания данных.

Для обеспечения стабильной нагрузки на конструктор задания каждый хост-потребитель сервиса канала имеет собственный пул потоков. Каждый поток потребляет и обрабатывает по одному событию. Бэкенд и каждый сервис канала регистрируют свои запросы для таких целей, как диагностика и аудит.

9.4. ХРАНИЛИЩЕ ОБЪЕКТОВ: КОНФИГУРАЦИЯ И ОТПРАВКА УВЕДОМЛЕНИЙ

Сервис уведомлений направляет поток событий сервисам каналов: каждое событие соответствует одной задаче уведомления для одного адресата.

> **ВОПРОС** Что, если уведомление содержит большие файлы или объекты? Добавлять один большой файл/объект в несколько событий Kafka неэффективно.

На рис. 9.3 бэкенд отправляет в топик Kafka уведомление размером 1 Мбайт. Однако оно может содержать большие файлы или объекты. Например, телефонное уведомление может содержать большой аудиофайл, а уведомление по электронной почте — несколько вложений в формате видео. Бэкенд сначала отправляет эти большие объекты в запросе POST в хранилище объектов, которое возвращает идентификаторы объектов. Затем бэкенд генерирует событие уведомления, которое включает эти идентификаторы объектов вместо исходных объектов, и отправляет это событие в соответствующий топик Kafka. Сервис канала потребляет событие, читает объекты из хранилища объектов запросом GET, конструирует уведомление, а затем доставляет его получателю. На рис. 9.4 в высокоуровневую архитектуру добавлен сервис метаданных.

Если отдельный большой объект доставляется нескольким получателям, бэкенд отправит в хранилище объектов несколько запросов POST. Начиная со второго запроса POST, хранилище объектов будет возвращать ответ 304 («Не изменялось»).

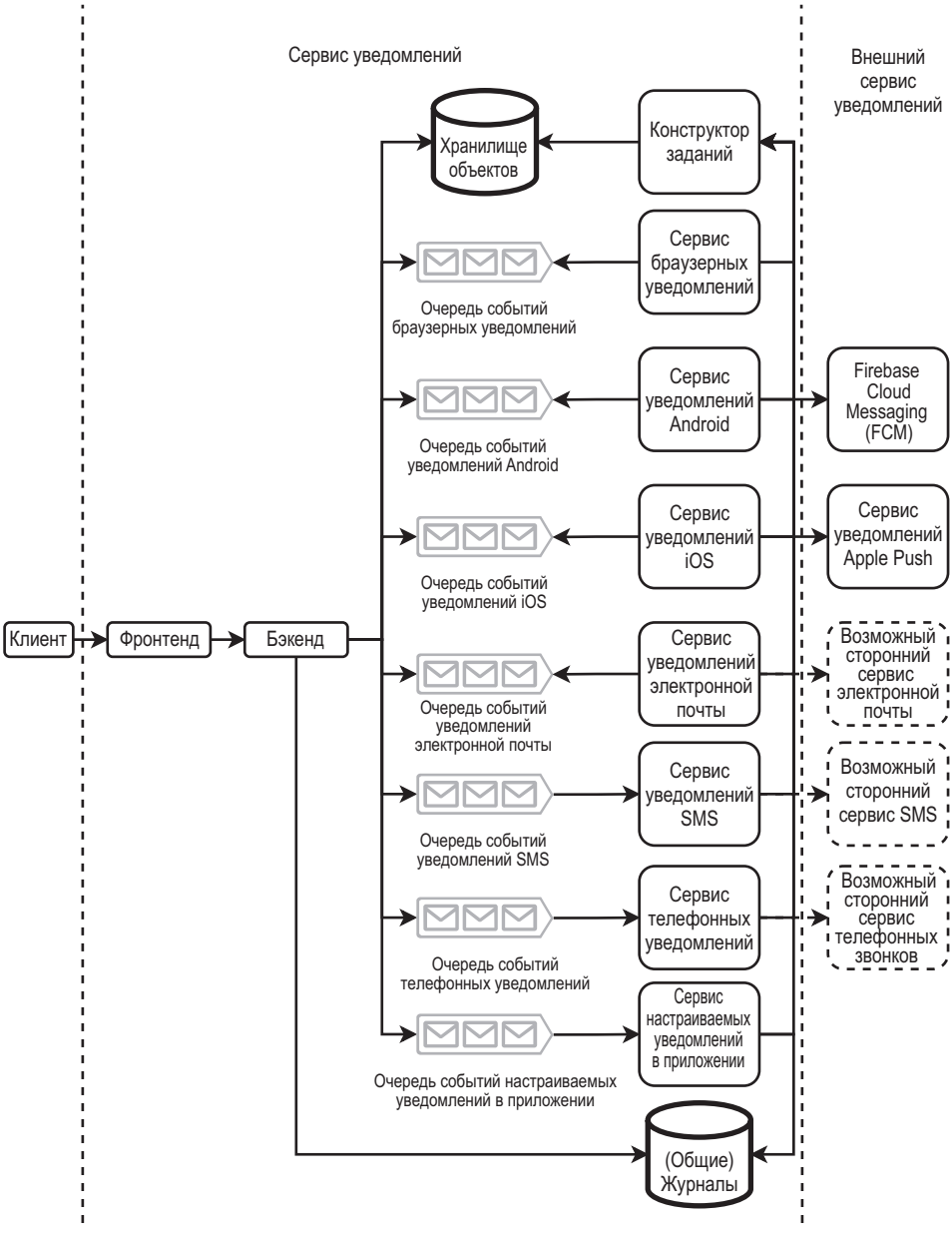

Рис. 9.4. Высокоуровневая архитектура с сервисом метаданных. Сервис бэкенда отправляет крупные объекты в сервис метаданных в запросе POST, чтобы события уведомлений оставались небольшими

9.5. ШАБЛОНЫ УВЕДОМЛЕНИЙ

Группа адресатов с миллионами приемников может привести к генерированию миллионов событий. Это вызывает значительные затраты памяти Kafka. В предыдущем разделе мы обсудили, как использовать сервис метаданных для сокращения дублирования контента в событиях и, как следствие, уменьшения их размеров.

9.5.1. Сервис шаблонов уведомлений

Многие события уведомлений представляют собой практически полные дубликаты с небольшими персонализированными изменениями. Например, на рис. 9.5 показано уведомление, которое может рассылаться миллионам пользователей. Оно содержит изображение, общее для всех получателей, и строку, в которой изменяется только имя получателя. Другой пример — отправка электронного письма, в котором большая часть содержимого будет одинаковой для всех пользователей. Заголовок и тело сообщения могут лишь незначительно различаться для каждого получателя (например, именами или процентом скидки), тогда как вложения, скорее всего, будут одинаковыми для всех получателей.

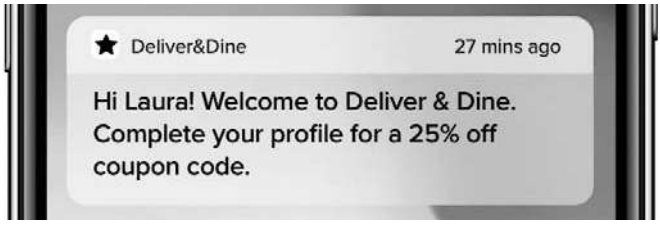

Рис. 9.5. Пример уведомления с изображением, общим для всех пользователей, и строкой, в которой изменяется только имя получателя. Общий контент помещен в шаблон вида «Hi ${name}! Welcome to Deliver & Dine». Событие в очереди Kafka может содержать пару «ключ — значение» в форме («name» и имя получателя, идентификатор приемника). Изображение взято из https://buildire.com/what-is-a -push-notiication/

В разделе 9.1.4 мы обсуждали, что шаблоны упрощают подобную персонализацию. Шаблоны также улучшают масштабирование сервиса уведомлений. Размещение всех общих данных в шаблоне сводит к минимуму размеры событий уведомлений. Создание и управление шаблонами само по себе может представлять сложную систему. Ее можно назвать сервисом шаблонов уведомлений, или, сокращенно, сервисом шаблонов. На рис. 9.6 изображена высокоуровневая архитектура сервиса шаблонов. Клиенту достаточно включить в уведомление идентификатор шаблона, и сервис канала с помощью запроса GET получит шаблон от сервиса шаблонов при генерировании уведомления.

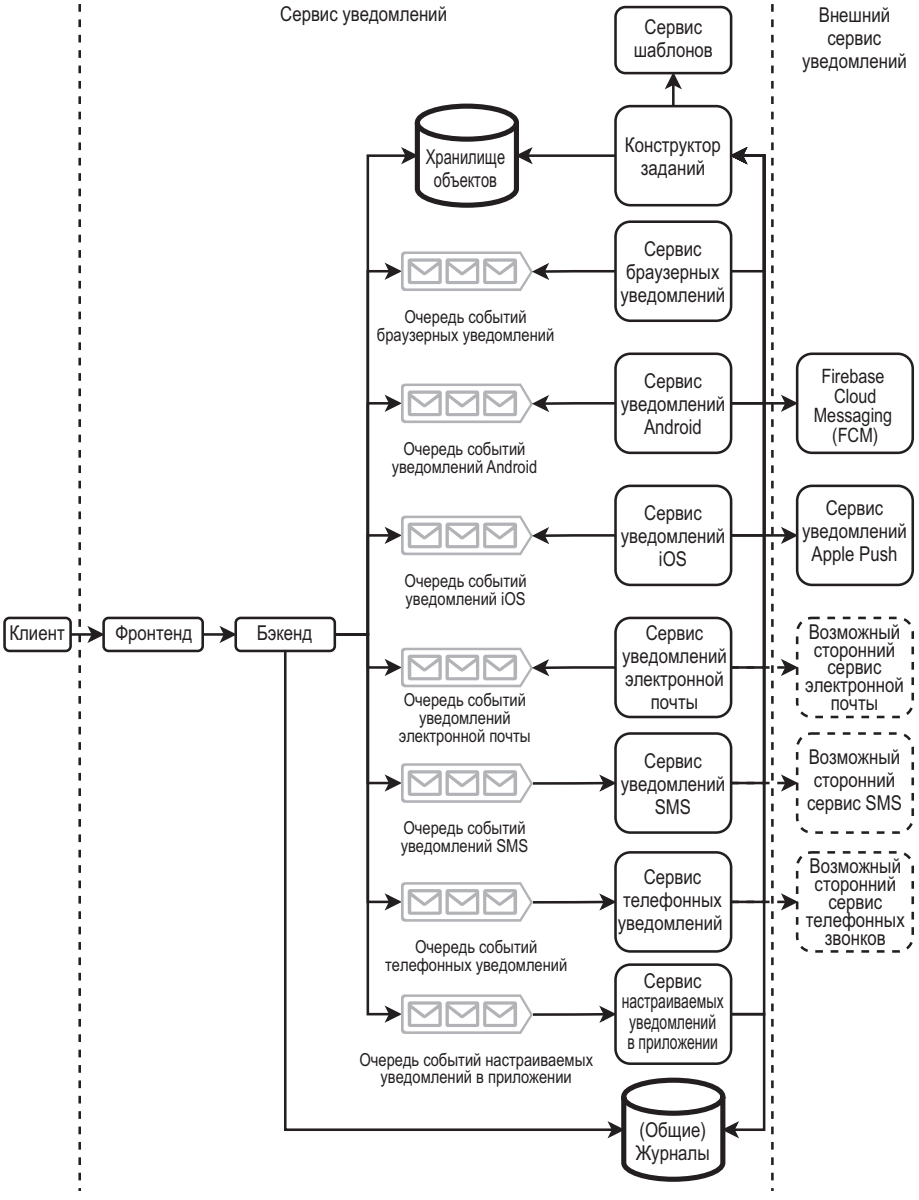

Рис. 9.6. Высокоуровневая архитектура с сервисом шаблонов. Пользователи сервисов уведомлений могут работать с шаблонами с использованием операций CRUD. Сервис шаблонов должен иметь собственные механизмы аутентификации, авторизации и RBAC (управления доступом на основе ролей). Конструктор заданий должен обладать доступом только для чтения. Администраторы должны обладать административным доступом, чтобы создавать, обновлять и удалять шаблоны или назначать роли другим пользователям

Объединяя этот подход с сервисом метаданных, мы приходим к тому, что событие должно содержать только идентификатор уведомлений (который также может использоваться как ключ шаблона уведомления), любые персонализированные данные в форме пар «ключ — значение» и приемник. Если в уведомлении отсутствует персонализированный контент (то есть оно одинаково для всех получателей), сервис метаданных содержит фактически весь контент уведомлений, а событие содержит только приемник и идентификатор контента уведомлений.

Пользователь создает шаблон уведомления до отправки уведомлений. Пользователь отправляет запросы CRUD или шаблоны уведомлений уровню сервиса, который перенаправляет их сервису метаданных для выполнения соответствующих запросов к базе данных, содержащей метаданные. В зависимости от доступных ресурсов или соображений простоты использования также можно разрешить пользователям не создавать шаблон и просто отправлять полные события уведомлений в сервис.

9.5.2. Дополнительные возможности

Вы можете добавить в шаблоны дополнительные возможности, такие как перечисленные ниже. Их можно кратко обсудить ближе к концу собеседования. Вряд ли у вас будет время, чтобы подробно на них останавливаться. Признаком инженерной зрелости и хорошим сигналом для эксперта является умение соискателя предвидеть такие возможности с одновременной демонстрацией способности быстро менять уровень детализации рассмотрения любой из этих систем и четко и лаконично их описывать.

Создание, управление доступом и управление изменениями

Пользователь должен иметь возможность создавать шаблоны. Система должна хранить данные шаблона, включая его контент и подробную информацию (идентификатор автора, метки времени создания и обновления и т. д.).

Для пользователей определены следующие роли: администратор, запись, чтение и запрет доступа. Они соответствуют разрешениям на доступ пользователя к шаблону. Возможно, сервис шаблонов уведомлений необходимо интегрировать с сервисом управления пользователем организации, который использует протокол LDAP.

Возможно, также стоит хранить историю изменений шаблонов, включая такие данные, как внесенные изменения, пользователь, который их внес, и метка времени. Можно пойти еще дальше и подумать о разработке процесса подтверждения изменений. Изменения, внесенные некоторыми ролями, могут требовать утверждения одного или нескольких администраторов. Такой подход можно обобщить до общего сервиса подтверждения, когда один или несколько пользователей предлагают операцию записи, а один или несколько пользователей подтверждают или запрещают операцию.

Тему управления изменениями можно продолжить: возможно, пользователь захочет отменить предыдущее изменение или вернуться к конкретной версии.

Повторно используемые и расширяемые функции и классы шаблонов

Шаблон может состоять из подшаблонов, пригодных для повторного использования, которые имеют разных владельцев и управляются независимо друг от друга. Будем называть их классами шаблонов.

Параметрами шаблонов могут быть переменные или функции. Функции полезны для реализации динамического поведения на устройстве получателя.

Переменная может иметь тип данных (`integer`, `varchar(255)` и т. д.). Когда клиент создает уведомление на основе шаблона, бэкенд проверяет значения параметров. Сервис уведомлений также может предоставить дополнительные ограничения/правила проверки (например, максимальное или минимальное целочисленное значение или длина строки). Правила проверки можно определить в функциях.

Параметры шаблонов могут заполняться простыми правилами (например, поле для имени получателя или поле для обозначения денежной единицы) или моделями машинного обучения (например, разным пользователям могут предлагаться разные скидки). Это потребует интеграции с системами, которые поставляют данные, необходимые для добавления динамических параметров. Управление контентом и персонализация — разные функции, принадлежащие разным командам, а сервисы и их интерфейсы должны проектироваться так, чтобы они четко отражали эту иерархию и разделение обязанностей.

Поиск

Сервис шаблона может хранить множество шаблонов и классов шаблонов; некоторые из них могут быть одинаковыми или очень похожими. Возможно, вы захотите реализовать функциональность поиска. В разделе 2.6 обсуждается, как это сделать.

Другое

Перед вами открываются бесконечные возможности. Например, как организовать управление CSS и JavaScript в шаблонах?

9.6. ПЛАНИРОВАНИЕ УВЕДОМЛЕНИЙ

Сервис уведомлений может использовать общий сервис Airflow или сервис планировщика заданий для организации планирования уведомлений. На рис. 9.7 сервис бэкенда предоставляет конечную точку API для планирования уведомлений, а также генерирует и отправляет соответствующий запрос к сервису Airflow для создания запланированного уведомления.

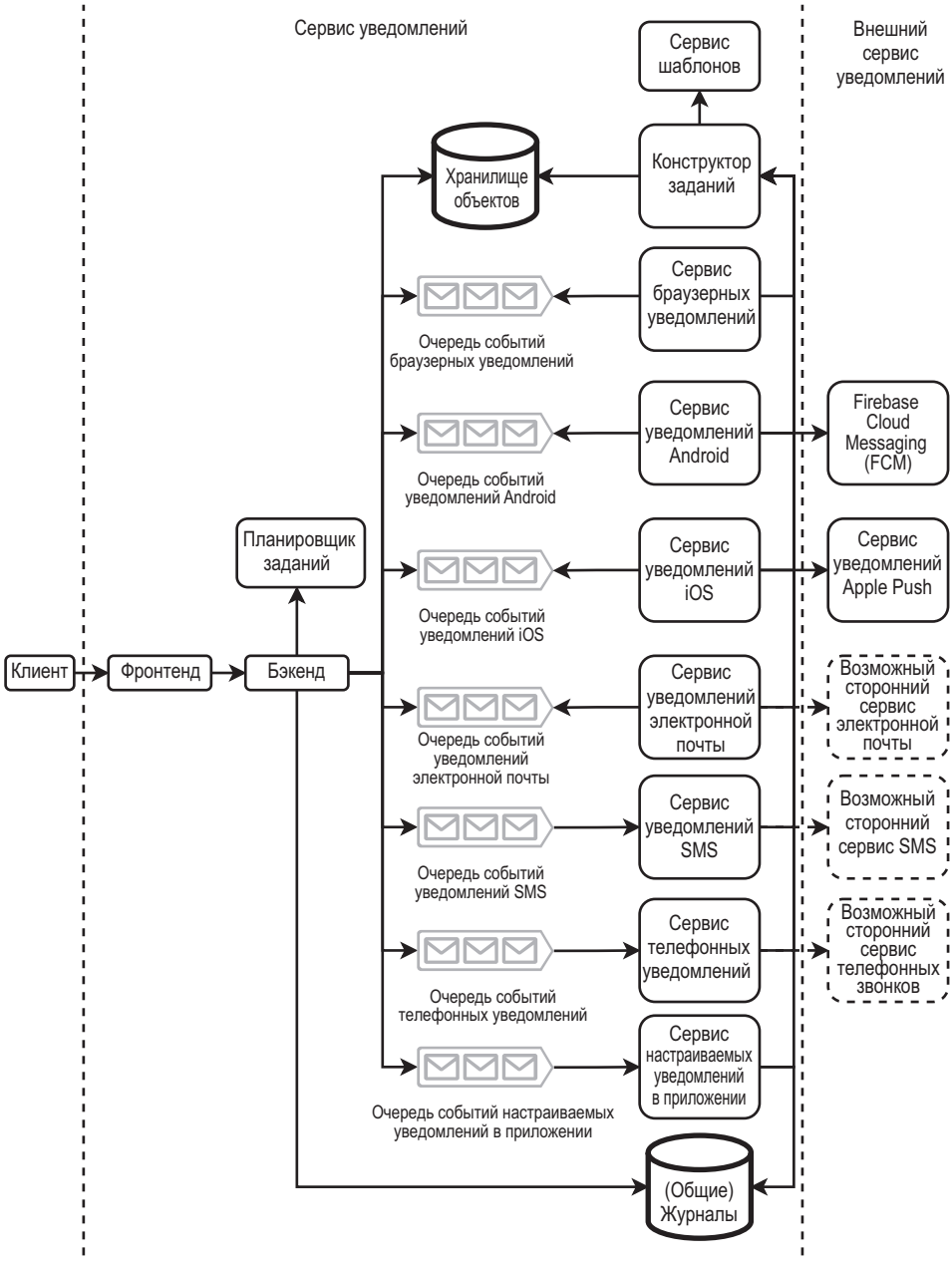

Рис. 9.7. Высокоуровневая архитектура с сервисом Airflow / планировщика заданий. Сервис планировщика заданий предназначен для настройки периодических уведомлений пользователями. В запланированное время сервис планировщика заданий отправляет события уведомлений на бэкенд

Когда пользователь создает или изменяет периодическое уведомление, скрипт Python задания Airflow генерируется автоматически и включается в репозиторий кода планировщика. Подробное обсуждение сервиса Airflow выходит за рамки книги. Для целей собеседования эксперт может предложить спроектировать собственную систему планирования задач вместо готового решения (такого, как Airflow или Luigi). Можно использовать решение на базе cron, описанное в разделе 4.6.1.

Периодические уведомления могут конкурировать с ситуационными, потому что на те и другие действуют ограничители частоты запросов. Все случаи, когда ограничитель частоты не позволяет запросу уведомления немедленно продолжить выполнение, необходимо регистрировать в журнале. В системе должен присутствовать дашборд, отображающий фактическую частоту событий, ограничиваемых по частоте. Также необходимо добавить оповещение, которое будет отправлено при высокой частоте ограничиваемых событий. На основании этой информации можно масштабировать размер кластера, выделять больше средств на внешние сервисы уведомлений, а также ограничивать некоторых пользователей от выдачи избыточных уведомлений.

9.7. ГРУППЫ АДРЕСАТОВ УВЕДОМЛЕНИЙ

Уведомление может иметь миллионы приемников/адресов. Если пользователь должен будет задавать всех получателей, ему придется поддерживать собственный список, что ведет к дублированию данных получателей между пользователями. Более того, передача миллионов приемников сервису уведомлений означает интенсивный сетевой трафик. Пользователям будет удобнее вести список приемников в сервисе уведомлений и использовать идентификатор из этого списка в запросах на отправку уведомлений. Будем называть такой список «группой адресатов уведомлений». Запрос на отправку уведомления может содержать либо список приемников (вплоть до установленного порога), либо список идентификаторов группы адресатов.

Для работы с этими группами можно спроектировать сервис групп адресов. Другие возможные функциональные требования этого сервиса:

- Управление доступом для различных ролей: только для чтения, только для добавления данных (можно добавлять, но нельзя удалять адреса), административный доступ (полный доступ). Управление доступом — важное средство безопасности, потому что неавторизованный пользователь может отправлять уведомления всей пользовательской базе с миллиардом получателей, а эти уведомления могут содержать спам или более вредоносную активность.
- Адресатам может быть разрешено удалять себя из групп уведомлений, чтобы избежать отправки спама. События удаления могут регистрироваться в журнале для аналитики.

- Доступ к функциональности может предоставляться через конечные точки API, и обращения ко всем этим точкам происходят на уровне сервисов.

Также может возникнуть необходимость в ручном рецензировании и утверждении запросов на уведомления к большому числу получателей. Уведомления в тестовых средах не требуют подтверждения, тогда как в рабочей среде их необходимо подтверждать вручную. Например, запрос уведомлений к одному миллиону получателей может потребовать ручного подтверждения со стороны группы эксплуатации, к 10 миллионам получателей — со стороны руководства, к 100 миллионам — старшего руководства, а уведомления для всей пользовательской базы потребуют подтверждения на уровне директора. Можно спроектировать систему, которая позволит отправителям получить подтверждение до отправки уведомлений. Но эту тему мы рассматривать не будем.

На рис. 9.8 изображена наша высокоуровневая архитектура с сервисом групп адресов. Пользователь указывает группу адресов в запросе на уведомление. Бэкенд отправляет запросы GET к сервису групп адресов для получения идентификаторов пользователей, входящих в группу адресов. Так как группа может содержать более миллиарда идентификаторов пользователей, один запрос GET может не включить все идентификаторы, а ограничиться максимальным количеством. Сервис адресов групп должен предоставить конечную точку `GET /address-group/count/{`*имя*`}`, которая возвращает количество адресов в группе, и конечную точку `GET /address-group/{`*имя*`}/start-index/{`*нач-индекс*`}/end-index/{`*кон-индекс*, чтобы бэкенд мог выдавать GET-запросы на получение пакетов адресов.

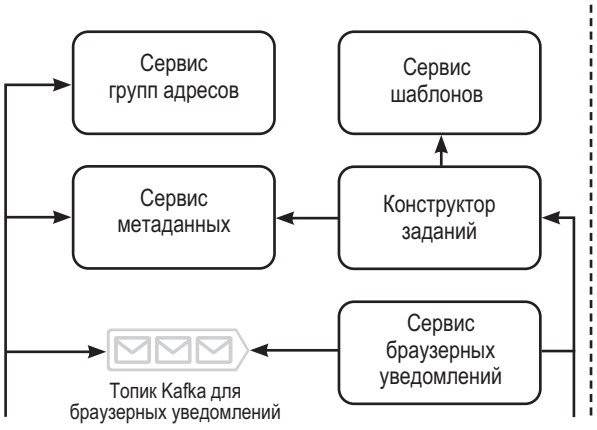

Рис. 9.8. Увеличенный фрагмент рис. 9.6 с добавлением сервиса групп адресов. Группа адресов содержит список получателей. Сервис группы адресов позволяет пользователю отправить уведомление нескольким пользователям, указав имя группы адресов вместо перечисления всех получателей

Можно воспользоваться хореографической сагой (раздел 5.6.1) для получения этих адресов и генерации событий уведомлений. Они могут справляться с выбросами трафика к сервису групп адресов. На рис. 9.9 показана архитектура бэкенда для решения этой задачи.

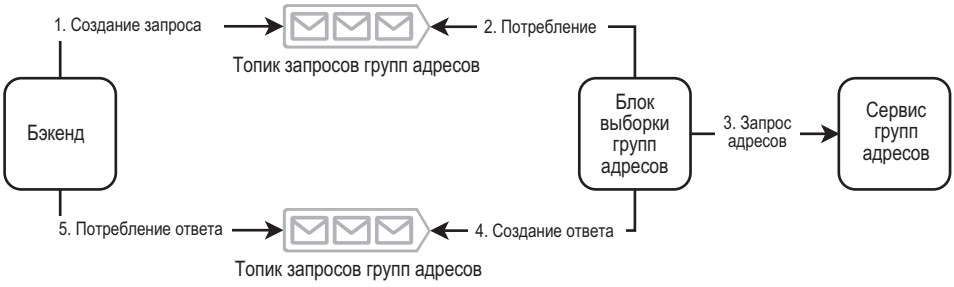

Рис. 9.9. Архитектура бэкенда для конструирования событий уведомлений для группы адресов

На диаграмме последовательности действий на рис. 9.10 производитель создает событие для такого задания. Потребитель потребляет это событие, после чего:

1. Использует GET для получения пакета адресов от сервиса групп адресов.
2. Генерирует событие уведомления для каждого адреса.
3. Отправляет его в соответствующий топик Kafka для событий уведомлений.

Стоит ли делить сервис бэкенда на две части, чтобы шаги 5 и далее выполнялись другим сервисом? Мы не сделали этого, потому что бэкенду не нужно отправлять запросы к сервису группы адресов.

> **СОВЕТ** Бэкенд отправляет запрос в один топик и потребляет из другого. Если вам нужна программа, которая потребляет из одного топика и отправляет в другой, обратите внимание на Kafka Streams (https://kafka.apache.org/10/documentation/streams/).

> **ВОПРОС** Что, если во время получения адресов блоком выборки группы адресов в группу адресов будут добавлены новые пользователи?

Проблема в том, что вы немедленно обнаружите, что большая группа адресов быстро изменяется. Новые получатели постоянно добавляются или удаляются из группы по разным причинам:

- У них могут изменяться телефоны или адреса электронной почты.
- В любое время в приложении могут как появляться новые пользователи, так и удаляться существующие.
- В случайной выборке, включающей миллиард человек, ежедневно рождаются и умирают тысячи людей.

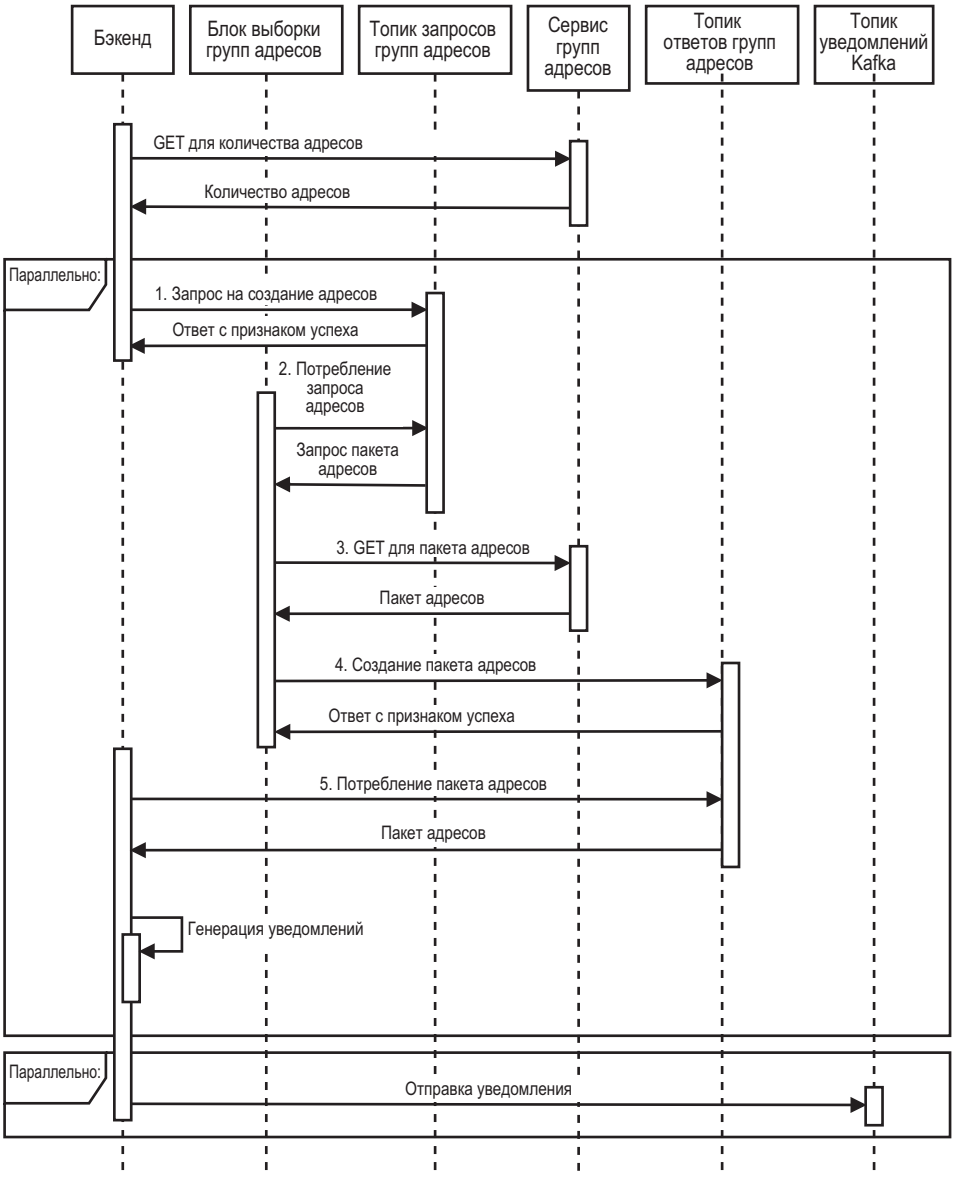

Рис. 9.10. Диаграмма последовательности действий в сервисе бэкенда по конструированию событий уведомлений для групп адресов

Когда уведомление считается доставленным всем получателям? Если бэкенд продолжит загружать пакеты новых получателей для создания событий уведомлений, то при достаточно большой группе создание событий никогда

не завершится. Уведомление необходимо доставлять только тем пользователям, которые находились в группе адресов в момент активации запроса на отправку.

В этой книге мы не будем касаться обсуждения возможных архитектур и подробностей реализации сервиса группы адресов.

9.8. ЗАПРОСЫ НА ОТМЕНУ ПОДПИСКИ

Каждое уведомление должно содержать кнопку, ссылку или другие элементы UI, при помощи которых получатели могут отменить подписку на похожие уведомления. Если получатель захочет отказаться от получения уведомлений в будущем, отправитель должен быть оповещен об этом.

Также можно добавить в приложение страницу управления уведомлениями для пользователей как на рис. 9.11, где пользователи приложения могут выбирать категории уведомлений, которые они желают получать. Сервис уведомлений должен предоставить список категорий уведомлений, а запрос уведомления должен содержать обязательное поле категории.

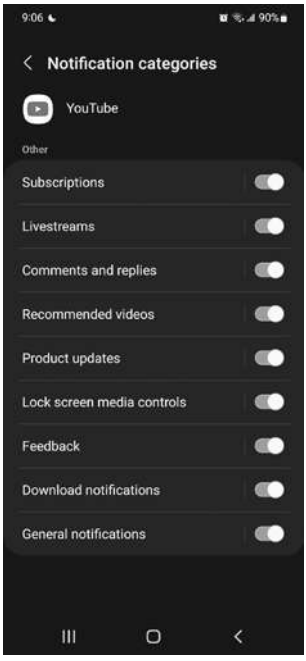

Рис. 9.11. Управление уведомлениями в приложении YouTube для Android. Вы можете определить список категорий уведомлений, чтобы пользователи приложений выбирали, на какие категории они хотят подписаться

ВОПРОС Где лучше реализовать отмену подписки — на стороне клиента или на стороне сервера?

Либо на стороне сервера, либо на обеих сторонах. Не реализуйте ее только на стороне клиента. Если отмена подписки реализована только на стороне клиента, сервис уведомлений будет продолжать отправлять уведомления получателям, а приложение на устройстве получателя будет их блокировать. Этот подход можно применить для браузерных и мобильных приложений, но его не удастся реализовать для электронной почты, телефонных звонков или SMS. Более того, генерация и отправка сообщений только для того, чтобы потом их заблокировал клиент, — напрасная трата ресурсов. Тем не менее блокировку уведомлений можно реализовать и на стороне клиента, если реализация на стороне сервера содержит ошибки и отправляет уведомления, которые должны быть заблокированы.

Если отмена подписки будет реализована на сервере, то сервис уведомлений будет блокировать отправку уведомлений получателю. Бэкенд должен предоставить конечную точку API для подписки или отмены подписки на уведомления, а кнопка/ссылка в интерфейсе должна отправлять запрос этому API. В одной из возможных реализаций блокировки уведомлений API сервиса групп адресов изменяется так, чтобы принимать категории. Новые конечные точки GET APT должны иметь вид `GET /address-group/count/{`*`имя`*`}/category/{`*`категория`*`}` и `GET /address-group/{`*`имя`*`}/category/{`*`категория`*`}/start-index/{`*`нач-индекс`*`}/end-index/{`*`кон-индекс`*. Сервис группы адресов будет возвращать только тех получателей, которые принимают уведомления из этой категории. Архитектура и подробности реализации выходят за рамки этого обсуждения.

9.9. ОШИБКИ ПРИ ДОСТАВКЕ

Доставка уведомлений может завершиться неудачей по причинам, не связанным с самим сервисом уведомлений:

- Не удается связаться с устройством получателя по таким причинам, как:
  - проблемы с сетью;
  - устройство получателя выключено;
  - сторонние сервисы доставки недоступны;
  - пользователь удалил мобильное приложение или свою учетную запись. При удалении учетной записи или мобильного приложения должны существовать механизмы обновления сервиса группы адресов, но обновление еще не было применено. Сервис канала может просто отбросить запрос и больше ничего не делать. Можно предположить, что сервис групп адресов будет обновлен в будущем, и тогда запросы GET от группы адресов не будут включать этого получателя.
- Получатель заблокировал эту категорию уведомлений, а устройство получателя заблокировало это уведомление. Уведомление было доставлено, хотя и не должно было (вероятно, из-за ошибки). Для таких случаев следует настроить оповещение с низкой степенью критичности.

Каждый случай, упомянутый в подпунктах первого пункта должен обрабатываться по своим правилам. Возникновение сетевых проблем, влияющих на работу вашего датацентра, крайне маловероятно. Если оно все же произошло, ответственная команда должна была уже разослать оповещение всем командам, кого это могло затронуть (очевидно, по каналам, не зависящим от датацентра). Вы вряд ли будете подробно обсуждать эту тему на собеседовании.

При возникновении сетевых проблем, которые влияют только на конкретного получателя, или в случае, если устройство получателя отключено, сторонний сервис доставки вернет вашему сервису канала ответ с этой информацией. Сервис канала может добавить в событие уведомления счетчик повторных попыток или же увеличить счетчик, если поле повторных попыток уже присутствует (то есть доставка уже была повторной попыткой). Затем он отправляет это уведомление в топик Kafka, который действует как очередь недоставленных сообщений. Сервис канала потребляет события из очереди недоставленных сообщений, а затем повторяет запрос на доставку. На рис. 9.12 в высокоуровневую архитектуру

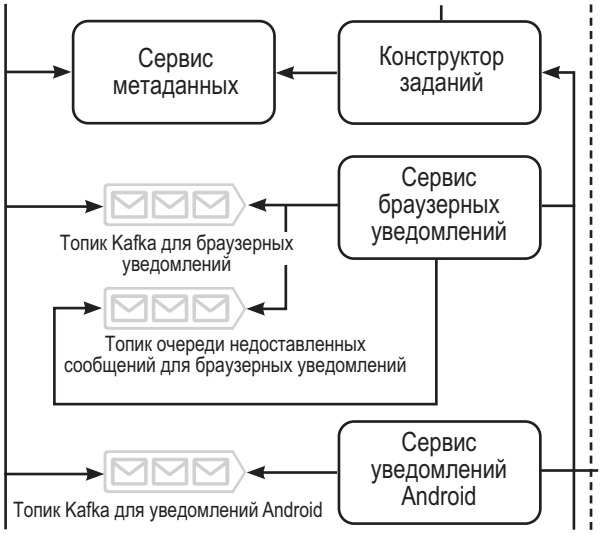

Рис. 9.12. Увеличенный фрагмент рис. 9.6 с добавлением очереди недоставленных сообщений браузерных уведомлений. Очереди недоставленных сообщений для других сервисов каналов будут выглядеть аналогично. Если сервис уведомления браузера сталкивается с ошибкой 503 (Сервис недоступен) при доставке уведомления, он ставит это событие уведомления в очередь недоставленных сообщений. Повторная попытка доставки будет выполнена позже. Если доставка завершается неудачей после трех попыток, сервис браузерных уведомлений регистрирует событие (в общем сервисе ведения журналов). Возможно, также стоит настроить оповещение с низкой степенью критичности для таких повторных попыток

добавлены очереди недоставленных сообщений. Если повторная попытка трижды завершается неудачей, сервис канала регистрирует информацию в журнале и отправляет сервису групп адресов запрос о невозможности связаться с пользователем. Сервис групп адресов должен предоставить для этой цели специальную конечную точку API, а также перестать включать этого пользователя в будущие запросы GET. Подробности реализации мы рассматривать не будем.

Если сторонний сервис доставки недоступен, сервис канала должен инициировать неотложное оповещение, применить экспоненциальную задержку и совершить повторную попытку с тем же событием. Сервис канала может увеличить интервал между повторными попытками.

Сервис уведомления также должен предоставить конечную точку API, через которую приложение-получатель сможет запрашивать пропущенные уведомления. Когда электронная почта, браузер или мобильное приложение получателя готовы к получению уведомлений, они отправляют запрос к конечной точке API.

9.10. ОСОБЕННОСТИ ДУБЛИРОВАНИЯ УВЕДОМЛЕНИЙ НА СТОРОНЕ КЛИЕНТА

Сервисы каналов, отправляющие уведомления непосредственно на устройства получателей, должны поддерживать как push-, так и pull-запросы. При создании уведомления сервис канала должен немедленно отправить его получателю. Однако клиентское устройство получателя может быть выключено или недоступно по иной причине. Когда устройство снова станет доступным, оно должно загрузить уведомления из сервиса уведомлений. Это относится к каналам, не использующим внешние сервисы уведомлений, например браузерам или приложениям с настраиваемыми уведомлениями.

Как избежать дублирования? Ранее мы рассмотрели некоторые решения, позволяющие предотвратить дублирование уведомлений от внешних сервисов (то есть push-запросов). Предотвращение дублирования уведомлений для pull-запросов должно быть реализовано на стороне клиента. Наш сервис не должен отклонять запросы к тем же уведомлениям (кроме, возможно, ограничения частоты), поскольку у клиента могут быть веские причины для повторения запросов. Клиент должен регистрировать уведомления, уже выведенные для пользователя (и закрытые им), возможно, в браузерном локальном хранилище или базе данных SQLite мобильного устройства. Когда клиент получает уведомление в pull- (и возможно, в push-) запросе, он должен провести поиск по хранилищу устройства, чтобы определить, отображалось ли уже оно, прежде чем отправлять новое уведомление пользователю.

9.11. ПРИОРИТЕТ

Уведомления могут иметь разные уровни приоритета. Руководствуясь схемой рис. 9.13, можно решить, сколько уровней приоритета нам понадобится (например, от 2 до 5), и создать отдельный топик Kafka для каждого уровня приоритета.

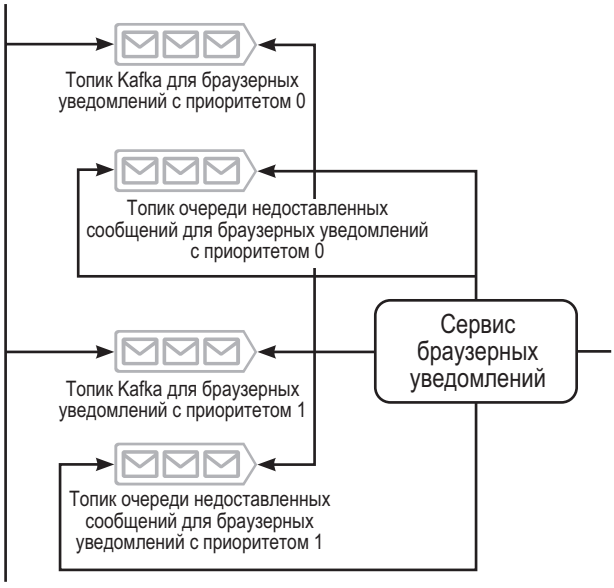

Рис. 9.13. Рисунок 9.12 с двумя уровнями приоритетов

Чтобы уведомления с более высоким приоритетом обрабатывались до низкоприоритетных, хост-потребитель может просто потреблять из высокоприоритетных топиков Kafka, пока они не опустеют, а затем переходить к потреблению из низкоприоритетных топиков. Также возможно решение с весами: готовый потреблять событие хост-потребитель выбирает подходящий топик Kafka, используя взвешенный случайный выбор.

ЗАДАНИЕ Расширьте дизайн системы, чтобы для разных каналов использовались разные конфигурации приоритетов.

9.12. ПОИСК

Вы можете реализовать систему поиска для нахождения и просмотра существующих конфигураций уведомлений/оповещений. Можно провести индексирование по шаблонам уведомлений и группам адресов уведомлений. Как было показано в разделе 2.6.1, для такого сценария использования будет достаточно поисковой фронтенд-библиотеки, например, `match-sorter`.

9.13. МОНИТОРИНГ И ОПОВЕЩЕНИЯ

Кроме того, что обсуждалось в разделе 2.5, можно реализовать мониторинг и оповещения по параметрам, перечисленным ниже.

Пользователи должны иметь возможность отслеживать состояние своих уведомлений. Эта возможность может предоставляться другим сервисом, который читает данные из сервиса ведения журналов. Вы можете предоставить UI сервиса уведомлений для пользователей, желающих создавать уведомления и управлять ими, включая шаблоны и отслеживание статусов уведомлений.

Также можно создать дашборды для ведения мониторинга статистических показателей. Кроме уже упомянутых частот успеха и неудачи, могут быть полезными такие показатели, как количество событий в очереди и процентили размеров событий по времени, с разбиением по каналам и приоритетам, а также такие параметры ОС, как потребление ресурсов процессора, памяти и дискового пространства. Высокое потребление памяти и большое количество событий в очереди указывают на чрезмерное потребление ресурсов; можно проанализировать события и определить, нельзя ли поместить часть данных в сервис метаданных для сокращения размеров событий в очереди.

Возможно проведение периодического аудита для выявления скрытых ошибок. Например, это можно сделать при помощи внешних сервисов уведомлений, которые будут сравнивать следующие два показателя:

- количество ответов 200, полученных сервисами уведомлений, отправляющими запросы к внешним сервисам уведомлений;
- количество действительных уведомлений, полученных этими внешними сервисами уведомлений.

Механизм выявления аномалий может использоваться для определения нетипичных изменений в частоте уведомлений или размере сообщений по разным параметрам, таким как получатель, отправитель или канал.

9.14. МОНИТОРИНГ ДОСТУПНОСТИ И УВЕДОМЛЕНИЯ/ОПОВЕЩЕНИЯ

В разделе 9.1.1 отмечалось, что мониторинг работоспособности нежелательно проводить с помощью сервиса уведомлений, поскольку он использует ту же инфраструктуру и сервисы, что и отслеживаемые сервисы. Но что, если сервис уведомлений должен использоваться как общий сервис оповещений об отказах в обязательном порядке? Что, если сам этот сервис выйдет из строя? Как сервис оповещений будет предупреждать пользователей? Одно из возможных

решений основано на использовании внешних устройств, например серверов, находящихся в разных датацентрах.

Можно предоставить клиентский демон, устанавливаемый на внешних устройствах. Сервис отправляет периодические управляющие сигналы внешним устройствам, настроенным на ожидание этих сигналов. Если устройство не получит сигнал в установленное время, оно обращается с запросом к сервису для проверки его работоспособности. Если система вернет ответ 2xx, устройство предполагает, что возникла временная проблема с сетью, и ничего не делает. Если при запросе происходит тайм-аут или возвращается ошибка, устройство оповещает пользователя(-ей) автоматизированным телефонным звонком, текстовым сообщением, электронным письмом, push-уведомлением и/или по другим каналам. По сути, это независимый, специализированный, мелкомасштабный сервис мониторинга и оповещений, который предназначен только для одной конкретной цели и отправляет оповещения только отдельным пользователям.

9.15. ДРУГИЕ ВОЗМОЖНЫЕ ТЕМЫ ОБСУЖДЕНИЙ

При необходимости также можно масштабировать (увеличить или уменьшить) объем памяти кластера Kafka. Если количество событий в очередях монотонно возрастает со временем, то уведомления не доставляются, и придется либо масштабировать потребительский кластер для обработки и доставки этих событий, либо реализовать ограничение частоты и оповещать соответствующих пользователей о превышении нагрузки.

Для общего сервиса можно рассмотреть возможность автоматического масштабирования. Тем не менее решения с автоматическим масштабированием непрактичны. На практике можно настроить автоматическое масштабирование для автоматического увеличения размеров кластеров разных компонентов сервиса до определенного порога, чтобы избежать сбоев при непредвиденных выбросах трафика; при этом также отправляются оповещения разработчикам для увеличения объема выделяемых ресурсов в случае необходимости. Также можно вручную проверить случаи срабатывания механизма автомасштабирования и соответствующим образом уточнить его конфигурации.

Подробное обсуждение сервиса уведомлений заняло бы целую книгу и включало бы множество общих сервисов. Чтобы сосредоточиться на основных компонентах сервиса уведомлений и сохранить объем материала в разумных пределах, в этой главе мы ограничились беглым упоминанием некоторых вопросов. В оставшееся время на собеседовании можно обсудить следующие темы:

- Получатель должен иметь возможность соглашаться на получение уведомлений и отказываться от него; без этого рассылка становится обычным спамом. Возможно, вы захотите уделить внимание этому вопросу.

- Как поступить, если необходимо исправить уведомление, уже разосланное большому количеству пользователей?
  - Если ошибка была обнаружена во время отправки уведомления, возможно, процесс стоит отменить и не отправлять уведомление остальным получателям.
  - Для устройств, на которых уведомления еще не сработали, можно отменить такие уведомления.
  - Для устройств, на которых уведомления уже сработали, следует отправить сопроводительное уведомление для объяснения ошибки.
- Вместо того чтобы ограничивать отправителя по частоте независимо от того, какие каналы он использует, спроектируйте систему, которая допускает ограничения частоты на отдельных каналах.
- Возможное применение аналитики:
  - Анализ времени доставки уведомлений по разным каналам, который может использоваться для повышения производительности.
  - Отслеживание частоты ответов на уведомления, аналитика пользовательских действий и других ответов на уведомления.
  - Интеграция системы оповещений с системой A/B-тестирования.
- API и архитектура для дополнительных функций сервисов шаблонов, о которых шла речь в разделе 9.5.2.
- Масштабируемый сервис планировщика заданий с высокой доступностью.
- Системный дизайн сервиса групп адресов для поддержки функций, рассмотренных в разделе 9.7. Дополнительно можно обсудить такие вопросы, как:
  - Какой механизм использовать для обработки запросов на отмену подписки — пакетный или потоковый?
  - Как вручную заново подписать получателя на уведомления?
  - Автоматическая повторная подписка получателя на уведомления, если устройство или учетная запись получателя обращаются с запросами к любому сервису вашей организации.
- Сервис подтверждений, получающий и отслеживающий подтверждения для отправки уведомлений большому количеству получателей. Можно расширить это обсуждение до проектирования механизмов, предотвращающих злоупотребления и отправку нежелательных уведомлений.
- Дополнительные подробности мониторинга и оповещений, включая примеры и тщательную проработку конкретных метрик и сигналов.
- Дополнительное обсуждение клиентского демона.
- Проектирование различных сервисов передачи сообщений (например, сервиса электронной почты, сервиса SMS, сервиса автоматизированных телефонных звонков и т. д.).

9.16. ЗАКЛЮЧИТЕЛЬНЫЕ ЗАМЕТКИ

Наше решение масштабируемо. Каждый компонент горизонтально масштабируем. Отказоустойчивость исключительно важна в этом общем сервисе, и мы постоянно обращали на нее внимание. Мониторинг и доступность спроектированы надежно; не существует единой точки сбоя, а подсистемы мониторинга и оповещения доступности и работоспособности системы используют независимые устройства.

ИТОГИ

- Сервис, который должен предоставлять одну функциональность на нескольких платформах, может состоять из одного бэкенда, централизующего общую обработку и передающего запросы соответствующему компоненту (или другому сервису) для каждой платформы.
- Используйте сервис метаданных и/или хранилище объектов для сокращения размера сообщений в очереди брокера сообщений.
- Автоматизируйте пользовательские действия с использованием шаблонов.
- Сервис планирования задач может использоваться для периодических уведомлений.
- Устранение дубликатов сообщений может выполняться на устройстве получателя.
- Взаимодействие между компонентами систем может осуществляться асинхронными средствами, такими как саги.
- Создайте дашборд с информацией для аналитики и отслеживания ошибок.
- Проводите периодический аудит и выявление аномалий для обнаружения возможных ошибок, которые не были зафиксированы в метриках.

10 Проектирование сервиса пакетного аудита базы данных

В ЭТОЙ ГЛАВЕ

- ✓ Аудит таблиц базы данных для выявления недопустимых данных
- ✓ Проектирование масштабируемого и точного решения для аудита таблиц базы данных
- ✓ Возможные способы решения нестандартных задач

Спроектируем общий сервис для ручной проверки данных. Это нестандартный вопрос на собеседованиях по проектированию систем — он не имеет однозначного решения, и подход, рассматриваемый в этой главе, — всего лишь один из многих.

Начнем эту главу со знакомства с концепцией *качества данных*. Существует много определений этого понятия. В общем случае под качеством данных понимается то, в какой степени набор данных подходит для выполнения своей задачи; также это может относиться к действиям, улучшающим пригодность набора данных для указанной цели. Качество данных определяется многими характеристиками. Следующие хаоактеристики взяты из https://www.heavy.ai/technical-glossary/data-quality:

- *Точность* (accuracy) — близость измеренного значения к истинному.
- *Полнота* (completeness) — данные содержат все обязательные значения для этой цели.

- *Консистентность*, или *согласованность* (consistency), — данные из разных расположений имеют одинаковые значения, а расположения проводят одинаковые изменения данных в одно время.
- *Допустимость* (validity) — данные отформатированы надлежащим образом, и их значения лежат в допустимом диапазоне.
- *Уникальность* (uniqueness) — отсутствие дублирования или наложения данных.
- *Своевременность* (timeliness) — данные доступны, когда в них возникает необходимость.

Существуют два подхода к проверке качества данных: выявление аномалий, рассмотренное в разделе 2.5.6, и ручная валидация (проверки данных, задаваемые вручную). В этой главе будет обсуждаться только ручная валидация. Например, таблица может обновляться ежечасно, но иногда обновления отсутствуют по несколько часов, при этом крайне маловероятно, что интервал между двумя обновлениями превысит 24 часа. Условие проверки данных выглядит так: «Последняя метка времени получена менее 24 часов назад».

Пакетный аудит с ручной валидацией — распространенное требование к системе. Супервизор транзакций (раздел 5.5) — один из многих возможных практических сценариев, хотя он не только проверяет, действительны ли данные, но и возвращает различия в данных между несколькими сервисами/базами данных, которые он сравнивает, вместе с операциями, необходимыми для восстановления согласованности между этими сервисами/базами данных.

10.1. ЗАЧЕМ НУЖЕН АУДИТ

Первое, что приходит в голову, — этот вопрос не имеет смысла. Однако без супервизора транзакции пакетный аудит может быть чреват закреплением плохих практик.

Например, если существует риск, что данные могут стать недействительными из-за потери данных в базах данных или файловых системах, следует реализовать репликацию или резервное копирование для предотвращения потери данных. Однако репликация или резервное копирование занимают какое-то время, а на хосте-лидере может произойти сбой до того, как данные будут успешно реплицированы или скопированы.

Если хост-лидер возобновит нормальную работу, данные можно восстановить, а затем реплицировать на других хостах. Однако это не сработает с некоторыми базами данных: например, база данных MongoDB в зависимости от конфигурации может намеренно удалять данные на хосте-лидере для поддержания согласованности (Артур Эйсмонт, «Web Scalability for Startup Engineers», McGraw Hill Education, 2015, p. 198–199). В базе данных MongoDB, если параметру Write

Concern (https://www.mongodb.com/docs/manual/core/replica-set-write-concern/) присвоено значение 1, все узлы должны быть согласованы с узлом-лидером. Если запись в узел-лидер будет успешной, но на нем произойдет сбой до выполнения репликации, остальные узлы используют консенсусный протокол для выбора нового лидера. Если первый лидер восстановится, он отменит все последние изменения, включая эту запись, чтобы данные не отличались от данных на новом узле-лидере.

Предотвращение потери данных

Одним из методов предотвращения потерь данных из-за запоздалой репликации является кворумная согласованность (то есть запись на большинство хостов/узлов кластера перед возвращением успешного ответа клиенту). В Cassandra запись реплицируется в структуре данных в памяти, называемой Memtable, на нескольких узлах перед возвращением успешного ответа. Запись в память также выполняется намного быстрее, чем запись на диск. Memtable записывается в структуру, называемую SSTable, либо периодически, либо по достижении определенного размера (например, 4 Мбайт).

Можно возразить, что данные должны проверяться во время их получения сервисом, а не после их сохранения в базе данных или файле. Например, когда сервис получает недопустимые данные, он должен возвращать ответ 4xx без сохранения этих данных. Следующие коды 4xx возвращаются для запросов записи с недопустимыми данными. За дополнительной информацией обращайтесь к таким источникам, как https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#client_error_responses:

- 400 Bad Request («Неправильный запрос») — универсальный ответ для любых запросов, которые сервис считает недопустимыми.
- 409 Conflict («Конфликт») — запрос конфликтует с правилом — например, если отправленная версия файла старше существующей на сервере.
- 422 Unprocessable Entity («Необрабатываемая сущность») — синтаксис запроса верный, но он не может быть обработан. Пример — запрос POST с телом JSON, содержащим недопустимые поля.

Существует еще один аргумент против аудита: проверка должна выполняться в базе данных и в приложении, а не во внешнем процессе аудита. Можно по максимуму использовать ограничения баз данных, потому что приложения изменяются намного быстрее, чем базы данных. Код приложения изменить проще, чем схему базы данных (провести миграцию базы данных). На уровне приложения оно само должно проверять входные и выходные данные, и в нем должны присутствовать модульные тесты для функций проверки ввода и вывода.

Ограничения базы данных

Существует мнение, что ограничения баз данных вредны (https://dev.to/jonlauridsen/database-constraints-considered-harmful-38), что они представляют собой преждевременную оптимизацию, не отражают все требования к целостности данных, усложняют тестирование системы и ее адаптацию к изменениям требований. Такие компании, как GitHub (https://github.com/github/gh-ost/issues/331#issuecomment-266027731) и Alibaba (https://github.com/alibaba/Alibaba-Java-Coding-Guidelines#sql-rules), запрещают ограничения по внешнему ключу.

На практике в приложении всегда будут ошибки, в том числе скрытые. Пример, который довелось отлаживать лично автору: в теле JSON конечной точки POST присутствовало поле даты, в котором должно было храниться значение будущей даты. Запросы POST проверялись, а затем записывались в таблицу SQL. Также существовало ежедневное пакетное задание ETL, которое обрабатывало объекты, помеченные текущей датой. Каждый раз, когда клиент отправлял запрос POST, бэкенд проверял, что значение даты в клиенте имеет верный формат и отстоит на одну неделю в будущем.

Однако таблица SQL содержала строку с датой, отстоящей на 5 лет в будущем, и этот факт оставался незамеченным в течение 5 лет, пока ежедневное пакетное задание ETL не обработало его и в конце серии пайплайнов ETL не был обнаружен недопустимый результат. Инженеры, написавшие этот код, ушли из компании, что усложняло отладку. Автор проверил историю git и обнаружил, что правило одной недели было реализовано только через несколько месяцев после того, как API был развернут в рабочей среде; был сделан вывод, что неверная строка данных была записана недопустимым запросом POST. Проверить предположение было невозможно из-за отсутствия журналов для запроса POST, потому что они хранились всего две недели. Периодическое задание аудита в таблице SQL обнаружило бы эту ошибку, даже если бы оно было реализовано и запущено спустя долгое время после записи данных.

Как бы мы ни старались избежать сохранения неверных данных, следует исходить из того, что ошибки неизбежны, и быть готовым к этому. Аудит — еще один уровень проверки данных.

Типичный сценарий применения пакетного аудита — проверка больших (например, > 1 Гбайт) файлов, особенно сгенерированных за пределами вашей организации и без вашего контроля. Обрабатывать и проверять каждую строку на одном хосте слишком долго. Если сохранить данные в таблице MySQL, можно воспользоваться командой `LOAD DATA` (https://dev.mysql.com/doc/refman/8.0/en/load-data.html), которая работает намного быстрее `INSERT`, а затем выполнить инструкцию `SELECT` для аудита данных. Инструкция `SELECT` намного быстрее и, пожалуй, проще запуска скрипта с файлом, особенно если воспользоваться индексами. Если вы работаете с распределенной файловой системой (такой, как

HDFS), подойдут альтернативные решения NoSQL, например Hive или Spark с быстрой параллельной обработкой.

Даже если будут найдены недопустимые значения, можно принять, что плохие данные лучше, чем их отсутствие, и все равно сохранить их в таблице.

Наконец, существуют проблемы, которые можно обнаружить только при пакетном аудите, например дублирование или отсутствие данных. Некоторые проверки данных могут требовать уже потребленных данных: например, алгоритмы обнаружения аномалий используют ранее потребленные данные для обработки и выявления аномалий в текущих потребляемых данных.

10.2. ВАЛИДАЦИЯ С УСЛОВИЕМ ПО РЕЗУЛЬТАТУ ЗАПРОСА SQL

Небольшое уточнение терминологии: таблица содержит строки и столбцы. Запись с определенными координатами (строка, столбец) может называться ячейкой, элементом, точкой данных или значением. В данной главе эти термины будут считаться синонимами.

Посмотрим, как задавать параметры ручной валидации при помощи операторов сравнения с результатами запроса SQL. Результат запроса SQL представляет собой двумерный массив, назовем его result. Для результата можно задать условие. Рассмотрим несколько примеров. Все проверки в них проводятся ежедневно, так что мы будем работать только со вчерашними строками, а запросы будут содержать условие `WHERE "Date(timestamp) > Curdate() - INTERVAL 1 DAY"`. В каждом примере описывается валидация, за которой следует запрос SQL и возможные условия.

Ручные валидации можно проводить для следующих сущностей:

- *Отдельные точки данных столбца.* Пример — возраст последней метки времени < 24 часов, что мы уже обсуждали.

```
SELECT COUNT(*) AS cnt
FROM Transactions
WHERE Date(timestamp) >= Curdate() - INTERVAL 1 DAY
```

- Возможные истинные условия — `result[0][0] > 0` и `result['cnt'][0] > 0`.
- Рассмотрим другой пример. Если идентификатор кода купона перестает действовать в определенную дату, можно задать для таблицы транзакций периодическую проверку, которая выдает оповещение, если идентификатор кода встречается после этой даты. Это может означать, что идентификаторы кодов купонов записаны неверно.

```
SELECT COUNT(*) AS cnt
FROM Transactions
```

```
WHERE code_id = @code_id AND Date(timestamp) > @date
➥AND Date(timestamp) = Curdate() - INTERVAL 1 DAY
```

- Возможные истинные условия — `result[0][0] == 0` и `result['cnt'][0] == 0`.
- *Несколько точек данных в столбце*. Например, если отдельный пользователь не может совершать более пяти покупок в день, можно задать для таблицы транзакций ежедневную проверку, которая выдает оповещение, если для одного идентификатора пользователя создано более пяти строк за период с предыдущего дня. Это может указывать на наличие ошибок, на то, что пользователю ошибочно разрешено совершить более пяти покупок в день, или на то, что покупки сохраняются некорректно.

```
SELECT user_id, count(*) AS cnt
FROM Transactions
WHERE Date(timestamp) = Curdate() - INTERVAL 1 DAY
GROUP BY user_id
```

- Условие — `result.length <= 5`.
- Другой вариант:

```
SELECT *
FROM (
  SELECT user_id, count(*) AS cnt
FROM Transactions
  WHERE Date(timestamp) = Curdate() - INTERVAL 1 DAY
  GROUP BY user_id
) AS yesterday_user_counts
WHERE cnt > 5;
```

- Условие — `result.length == 0`.
- *Несколько столбцов одной строки*. Например, общее количество продаж по одному коду купона не может превышать 100 в день.

```
SELECT count(*) AS cnt
FROM Transactions
WHERE Date(timestamp) = Curdate() - INTERVAL 1 DAY
➥AND coupon_code = @coupon_code
```

- Условие — `result.length <= 100`.
- Альтернативный запрос и условие:

```
SELECT *
FROM (
  SELECT count(*) AS cnt
FROM Transactions
WHERE Date(timestamp) = Curdate() - INTERVAL 1 DAY
➥AND coupon_code = @coupon_code
) AS yesterday_user_counts
WHERE cnt > 100;
```

Условие — `result.length == 0`.

- *Несколько таблиц.* Например, если имеется таблица фактов `sales_na`, хранящая информацию о продажах в Северной Америке, со столбцом `country_code`, можно создать таблицу измерения `country_codes` со списком кодов стран для каждого географического региона. Можно задать периодическую проверку того, что все новые строки содержат значения `country_code`, относящиеся к странам Северной Америки:

```
SELECT *
FROM sales_na S JOIN country_codes C ON S.country_code = C.id
WHERE C.region != 'NA';
```

  Условие — `result.length == 0`.

- *Условная команда для нескольких запросов.* Например, приложение отправляет оповещение, если количество продаж за день более чем на 10 % превышает продажи за тот же день на прошлой неделе. Также можно выполнить два запроса и сравнить их результаты, как описано ниже. Результаты запроса присоединяются к массиву результатов, так что массив результатов будет трехмерным, а не двумерным:

```
SELECT COUNT(*)
FROM sales
WHERE Date(timestamp) = Curdate()
SELECT COUNT(*)
FROM sales
WHERE Date(timestamp) = Curdate() - INTERVAL 7 DAY
```

Условие — `Math.abs(result[0][0][0] – result[1][0][0]) / result[0][0][0] < 0.1`.

Других сценариев для ручной валидации множество, включая следующие:

- минимальное количество новых строк, которые могут быть записаны в таблицу за час;
- заданный текстовый столбец не может содержать null-значения, а длина строки должна лежать в диапазоне от 1 до 255;
- заданный текстовый столбец должен содержать значения, совпадающие по заданному регулярному выражению;
- заданный целочисленный столбец содержит неотрицательное значение.

Некоторые виды ограничений также могут быть реализованы в виде аннотаций функций в библиотеках ORM (например, `@NotNull` и `@Length(min = 0, max = 255)`) в Hibernate или типами ограничений в пакете SQL в Golang. В таком случае сервис аудита служит дополнительным уровнем проверки. Неудачный аудит указывает на скрытые ошибки в создаваемом сервисе, которые необходимо изучить.

Примеры в этом разделе были основаны на SQL. Но рассматриваемую концепцию можно обобщить для написания запросов проверок на других языках, таких как

HiveQL, Trino (ранее известном как PrestoSQL) или Spark. Хотя наш дизайн ориентирован на использование языков запросов баз данных, функции проверки можно писать и на языках программирования общего назначения.

10.3. ПРОСТОЙ ПАКЕТНЫЙ СЕРВИС АУДИТА ДЛЯ ТАБЛИЦ SQL

В этом разделе мы сначала рассмотрим простой скрипт для аудита таблиц SQL. Затем обсудим, как создать задание пакетного аудита на базе этого скрипта.

10.3.1. Скрипт аудита

Простейшим вариантом задания пакетного аудита является скрипт, который выполняет следующие операции:

1. Выполнение запроса к базе данных.
2. Чтение результата в переменную.
3. Проверка значения переменной по некоторым условиям.

Скрипт Python из листинга 10.1 запускает запрос MySQL, который проверяет, что последняя метка времени таблицы транзакций не старше 24 часов, и выводит результат на консоль.

Листинг 10.1. Скрипт Python и запрос MySQL для проверки новейшей метки времени

```
import mysql
cnx = mysql.connector.connect(user='admin', password='password',
                          host='127.0.0.1',
                          database='transactions')
cursor = cnx.cursor()

query = """
SELECT COUNT(*) AS cnt
FROM Transactions
WHERE Date(timestamp) >= Curdate() - INTERVAL 1 DAY
"""

cursor.execute(query)
results = cursor.fetchall()
cursor.close()
cnx.close()

# result[0][0] > 0 - условие.
print(result[0][0] > 0) # result['cnt'][0] > 0 тоже работает.
```

Вероятно, вам придется выполнить несколько запросов к базе данных и сравнить их результаты. В листинге 10.2 приведен возможный пример.

Листинг 10.2. Пример скрипта для сравнения результатов нескольких запросов

```
import mysql

queries = [
    {
    'database': 'transactions',

   'query': """
       SELECT COUNT(*) AS cnt
        FROM Transactions
        WHERE Date(timestamp) >= Curdate() - INTERVAL 1 DAY
    """,
},
{
    `database': 'transactions`,
    'query': """
        SELECT COUNT(*) AS cnt
        FROM Transactions
        WHERE Date(timestamp) >= Curdate() - INTERVAL 1 DAY
        """
    }
]

results = []
for query in queries:
     cnx = mysql.connector.connect(user='admin', password='password',
                             host='127.0.0.1',
                             database=query['database'])
        cursor = cnx.cursor()
     cursor.execute(query['query'])
     results.append(cursor.fetchall())
cursor.close()
cnx.close()

print(result[0][0][0] > result[1][0][0])
```

10.3.2. Сервис аудита

Расширим концепцию до сервиса пакетного аудита. Скрипт можно обобщить так, чтобы пользователь мог задать:

- базы данных SQL и запросы;
- условие, которое должно применяться к результату запроса.

Реализуем шаблон в файле Python с именем validation.py.template. Возможная (упрощенная) реализация этого файла приведена в листинге 10.3. Задание пакетного аудита делится на два этапа:

1. Выполнение запросов к базе данных и определение того, успешно ли прошел аудит, на основании их результатов.
2. Если аудит завершился неудачей, выдается оповещение.

На практике регистрационные данные для входа будут предоставлены сервисом управления секретными данными, а хост — прочитан из файла конфигурации. Мы не будем обсуждать эти подробности. Практический сценарий для этого сервиса может выглядеть так:

1. Пользователь выполняет вход в сервис и создает новое задание пакетного аудита.
2. Пользователь вводит значения для базы данных, запросов и условия.
3. Сервис создает файл validation.py на базе validation.py.template и заменяет такие параметры, как `{database}`, значениями, введенными пользователем.
4. Сервис создает новое задание Airflow или cron, которое импортирует validation.py и запускает функцию проверки.

Нетрудно заметить, что файлы validation.py фактически представляют собой функции. Пакетный сервис ETL хранит функции вместо объектов.

Как указано в комментарии в файле validation.py.template, для каждого запроса базы данных следует создать задачу Airflow. Бэкенд должен сгенерировать такой файл validation.py. Эта задача отлично подойдет для оценки кандидата на вакансию программиста, но она выходит за рамки собеседования по системному проектированию.

Листинг 10.3. Шаблон файла Python для сервиса аудита

```
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.bash import BranchPythonOperator
import mysql.connector
import os
import pdpyras

# Примеры пользовательского ввода:
# {name} — ‘’
# {queries} — [‘’, ‘’]
# {condition} — result[0][0][0] result[1][0][0]

def _validation():
  results = []
  # Запросы к базам данных являются затратными операциями.
  # Проблема с выполнением всех запросов в этой точке заключается в том,
  # что в случае ошибки при выполнении одного запроса все запросы
  # придется выполнять заново. Вместо этого можно создать для каждого
  # запроса задачу Airflow.
  for query in {queries}:
    cnx = mysql.connector.connect(user='admin', password='password',
                                  host='127.0.0.1',
                                  database=query['database'])
    cursor = cnx.cursor()
    cursor.execute(query['query'])
```

```
  results.append(cursor.fetchall())
  cursor.close()
  cnx.close()
  # XCom - механизм Airflow для обмена данными между задачами.
  ti.xcom_push(key='validation_result_{name}', value={condition})

def _alert():
  # Примеры кода для выдачи оповещения PagerDuty в случае
  # неудачного аудита. Это всего лишь пример, который не следует
  # рассматривать как рабочий код.
  # Возможно, также стоит отправить этот результат сервису
  # бэкенда. Мы поговорим об этом позже в этой главе.
  result = ti.xcom_pull(key='validation_result_{name}')
  if result:
    routing_key = os.environ['PD_API_KEY']
    session = pdpyras.EventsAPISession(routing_key)
    dedup_key = session.trigger("{name} validation failed", "audit")

with DAG(
    {name},
    default_args={
       'depends_on_past': False,
       'email': ['zhiyong@beigel.com'],
       'email_on_failure': True,
       'email_on_retry': False,
       'retries': 1,
       'retry_delay': timedelta(minutes=5),
 },
 description={description},
 schedule_interval=timedelta(days=1),
 start_date=datetime(2023, 1, 1),
 catchup=False,
 tags=['validation', {name}],
) as dag:
 t1 = BranchPythonOperator(
     task_id='validation',
     python_callable=_validation
 )
 # Оповещение реализовано в отдельной задаче Airflow,
 # чтобы в случае неудачи при попытке оповещения задание
 # Airflow не выполняло заново затратную функцию проверки.
 t2 = BranchPythonOperator(
     task_id='alert',
     python_callable=_alert
 )
 t1 >> t2
```

10.4. ТРЕБОВАНИЯ

Спроектируем систему, в которой пользователи могут составлять запросы SQL, Hive или Trino (ранее Presto) для проведения периодических пакетных аудитов таблиц своих баз данных. Ниже перечислены функциональные требования.

- Задания аудита с поддержкой CRUD. Задание аудита содержит следующие поля:
  - интервал (минуты, часы, дни или нестандартные интервалы времени);
  - владельцы;
  - запрос проверки базы данных на SQL или родственных диалектах (HQL, Trino, Cassandra и т. д.);
  - условие с результатом запроса SQL.
- Задание с ошибкой должно выдавать оповещение.
- Просмотр журналов прошлых и текущих заданий, включая наличие ошибок и результаты выполнения условий. Пользователи также должны иметь возможность просматривать статус и историю всех выданных оповещений, например время выдачи и наличие пометки о завершении.
- Задание должно быть завершено не более чем за 6 часов.
- Запрос базы данных должен завершаться за 15 минут. Необоходим системный запрет заданий с длительными запросами.

Нефункциональные требования:

- *Масштаб* — предполагается, что заданий будет менее 10 000 (то есть 10 000 команд базы данных). Задания и их журналы читаются только через пользовательский интерфейс, так что трафик будет низким.
- *Доступность* — это внутренняя система, от которой другие системы напрямую не зависят. Высокая доступность не обязательна.
- *Безопасность* — на задания распространяется управление доступом. Операции CRUD могут выполняться с заданиями только их владельцами.
- *Точность* — результат задания аудита должен быть точным согласно определению в конфигурации задания.

10.5. ВЫСОКОУРОВНЕВАЯ АРХИТЕКТУРА

На рис. 10.1 изображена исходная высокоуровневая архитектура гипотетического сервиса, при помощи которого пользователи могут задавать периодические проверки своих таблиц. Предполагается, что пакетный сервис ETL является сервисом Airflow или работает аналогичным образом. Он сохраняет файлы Python пакетных заданий, запускает их по определенному расписанию, хранит статус и историю этих заданий и возвращает логические значения, указывающие, были ли условия аудита истинными или ложными. Пользователи взаимодействуют с интерфейсом, который отправляет запросы через бэкенд:

1. Пользователи могут отправлять запросы к общему пакетному сервису ETL на выполнение операций CRUD с заданиями пакетного аудита, включая проверку статуса и истории этих заданий.

2. Общий пакетный сервис ETL не является сервисом оповещений, поэтому в нем нет конечных точек API для выдачи оповещений или просмотра статуса и истории любых выданных оповещений. Чтобы просмотреть эту информацию, пользователи отправляют запросы к общему сервису оповещений через пользовательский интерфейс и бэкенд.

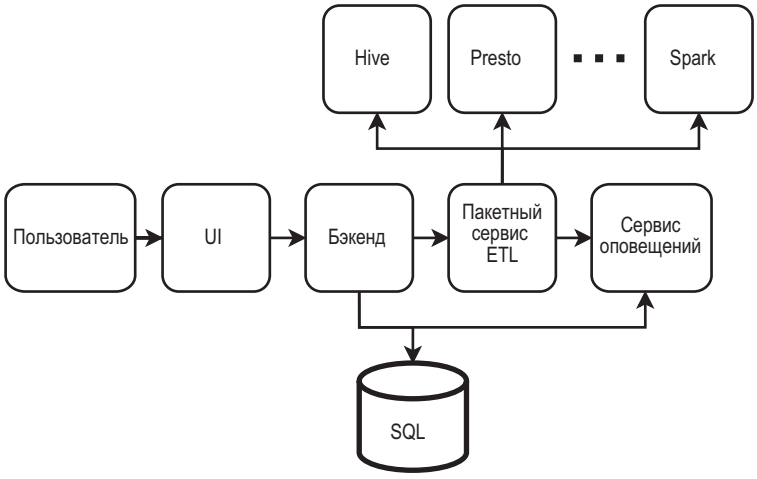

Рис. 10.1. Исходная высокоуровневая архитектура гипотетического сервиса, при помощи которого пользователь может задавать периодические проверки действительности данных

Когда пользователь отправляет запрос на создание пакетного задания аудита, выполняются следующие действия:

1. Сервис бэкенда создает файл validation.py, подставляя значения, введенные пользователем, в шаблон. Так как шаблон представляет собой короткую строку, он может быть сохранен в памяти на каждом хосте сервиса бэкенда.
2. Сервис бэкенда отправляет запрос пакетному сервису ETL с этим файлом. Пакетный сервис ETL создает пакетное задание ETL и сохраняет файл, после чего возвращает успешный ответ 200 сервису бэкенда.

Фактически сервис пакетного аудита представляет собой обертку для общего сервиса пакетных заданий ETL. Конфигурация задания аудита содержит такие поля, как владельцы задания, выражение cron, тип базы данных (Hive, Trino, Spark, SQL и т. д.) и выполняемый запрос. Главная таблица SQL содержит конфигурации заданий аудита, мы назовем ее `job_config`. Также можно создать таблицу `owner`, которая связывает задания с их владельцами и содержит столбцы `job_id` и `owner_id`.

Так как запросы проверки можно составлять на разных SQL-подобных диалектах, сервис пакетных заданий ETL может связываться с разными общими базами данных, такими как SQL, Hive, Trino, Spark, Cassandra и т. д. Если задание завершилось неудачей (или часть аудитов завершилась неудачей), сервис пакетных заданий ETL отправляет запросы к общему сервису для оповещения соответствующих специалистов. Для безопасности можно использовать общий сервис OpenID Connect для аутентификации, рассматриваемый в приложении Б.

10.5.1. Запуск задания пакетного аудита

Задание аудита периодически запускается с определенным интервалом и включает два основных этапа:

1. Выполнение запроса к базе данных.
2. Выполнение условной команды с результатом запроса базы данных.

Согласно сказанному в разделе 4.6.1, пакетное задание ETL создается как скрипт (например, скрипт Python в сервисе Airflow). Когда пользователь создает задание аудита, бэкенд генерирует соответствующий скрипт Python. Он может использовать заранее определенный и реализованный скрипт шаблона. Скрипт шаблона может содержать несколько секций, в которые подставляются соответствующие параметры (интервал, запрос к базе данных и условие).

Основные сложности масштабирования испытывает сервис пакетных заданий ETL и, возможно, сервис оповещений, так что обсуждение масштабирования сводится к проектированию масштабируемого сервиса пакетных заданий ETL и масштабируемого сервиса оповещений. Сервис оповещения подробно обсуждается в главе 9.

Так как пользовательское задание аудита в основном определяется как функция проверки, которая выполняет команду SQL, мы также предлагаем воспользоваться платформой FaaS (Function as a Service) и ее встроенным масштабированием. Также можно создать защиту от аномальных запросов: например, 15-минутное ограничение времени выполнения запроса или приостановку задания в случае недействительности результата запроса.

Результат каждого задания аудита может храниться в базе данных SQL, а пользователи будут обращаться к нему через предоставленный UI.

10.5.2. Обработка оповещений

Кто должен инициировать оповещения о неудачном аудите — сервис пакетных заданий ETL или бэкенд? Первая мысль — сервис пакетных заданий ETL: поскольку он запускает задания аудита, то должен и инициировать такие оповещения. Однако это означает, что функциональность оповещения, используемая сервисом пакетного аудита, распределяется между двумя компонентами:

- запросы на инициирование оповещений отправляются сервисом пакетных заданий ETL;
- запросы на просмотр статуса и истории оповещений отправляются сервисом бэкенда.

Это означает, что конфигурацию соединения с сервисом оповещений необходимо задавать в обоих сервисах, что создает дополнительные накладные расходы на обслуживание. В будущем в команду, занимающуюся обслуживанием сервиса пакетного аудита, могут прийти новые инженеры, незнакомые с кодом, и при возникновении проблем с оповещениями они могут сначала ошибочно предположить, что взаимодействия с сервисом оповещений выполняются в одном сервисе, и потратить лишнее время на отладку, прежде чем обнаружат, что проблема была в другом сервисе.

Таким образом, можно прийти к выводу, что все взаимодействие с сервисом оповещений должно осуществляться в сервисе бэкенда. Пакетное задание ETL будет только проверять, истинно или ложно условие, и отправлять это логическое значение сервису бэкенда. Если значение ложно, сервис бэкенда инициирует оповещение через сервис оповещений.

Однако такое решение может привести к ошибке. Если хост сервиса бэкенда, который генерирует и отправляет запросы оповещений, выйдет из строя или станет недоступен, оповещение может остаться неотправленным. Как избежать этой ошибки?

- Запрос от сервиса пакетных заданий ETL к сервису бэкенда можно сделать блокирующим, чтобы сервис бэкенда возвращал 200 только после успешной отправки запроса на оповещение. Чтобы убедиться, что запрос отправлен, можно воспользоваться механизмами повторных попыток сервиса пакетных заданий ETL (такими, как механизм повторных попыток в Airflow). Однако это будет означать, что сервис пакетных заданий ETL фактически отправляет запрос оповещения, что приводит к сильной связанности двух сервисов.
- Пакетный сервис ETL может отправлять события в секционированный топик Kafka, а хосты сервиса бэкенда могут потреблять из этих секций и создавать контрольные точки по каждой секции (возможно, с использованием SQL). Однако это может привести к дублированию оповещений, так как хост сервиса бэкенда может выйти из строя после выдачи запроса оповещения, но до создания контрольной точки. Сервис оповещений должен уметь удалять дубликаты оповещений.

В текущей архитектуре предусмотрено ведение журналов и мониторинг. Результаты аудита регистрируются в SQL, и осуществляется мониторинг заданий аудита; если при выполнении задания произошел сбой, сервис пакетного аудита выдает оповещение. Только оповещение выполняется общим сервисом.

Альтернативное решение — регистрация результатов задания аудита как в SQL, так и в общий сервис ведения журналов. Другая таблица SQL может использоваться с целью создания контрольных точек для определенного количества результатов.

Как видно на диаграмме последовательности действий на рис. 10.2, каждый раз, когда хост восстанавливается от сбоя, он обращается к таблице SQL для получения последней контрольной точки. Дубликаты журналов в SQL не создают проблем, поскольку можно просто использовать команды `INSERT INTO <table> IF NOT EXISTS`.... Есть три варианта действий с дубликатами в сервисе ведения журналов.

1. Принять, что последствия дублирования журналов незначительны, и просто записать их в сервис ведения журналов.
2. Обрабатывать дубликаты в сервисе ведения журналов.

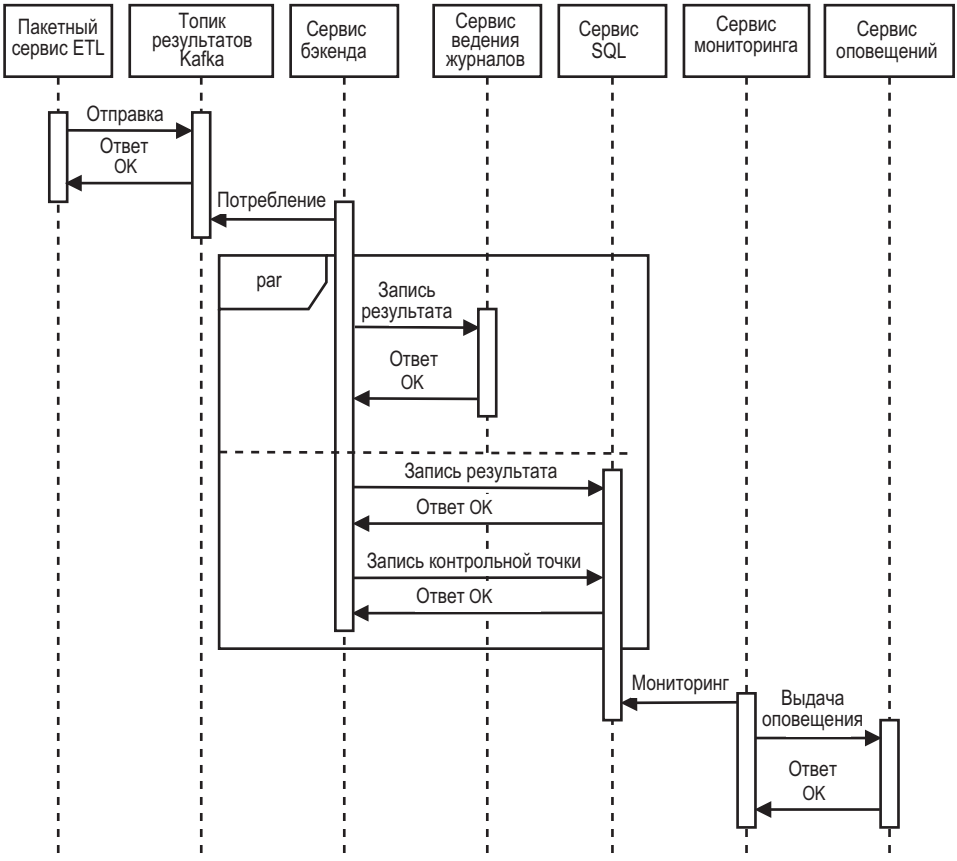

Рис. 10.2. Диаграмма последовательности действий демонстрирует параллельную запись в сервис ведения журналов и сервис SQL. Мониторинг и оповещения могут выполняться сервисом SQL

3. Прежде чем осуществлять запись, обратиться с запросом к сервису ведения журналов, чтобы определить, существует ли уже этот результат. При этом трафик к сервису ведения журналов удваивается.

На рис. 10.3 представлена обновленная высокоуровневая архитектура с общими сервисами ведения журналов и мониторинга. Ведение журналов и мониторинг слабо связаны с сервисом пакетных заданий ETL. Разработчики сервиса пакетных заданий ETL не зависят от изменений в сервисе оповещений, а сервис пакетных заданий ETL не нужно специально настраивать для отправки запросов к сервису оповещений.

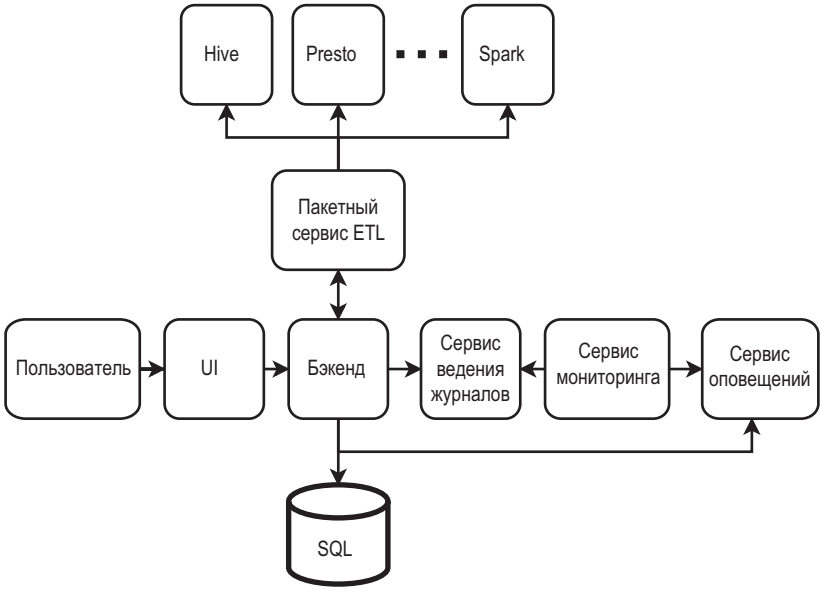

Рис. 10.3. Высокоуровневая архитектура с общими сервисами. Каждый сервис ведет запись в журнал через общий сервис ведения журналов, но на схеме показаны только его связи с сервисами бэкенда и мониторинга

10.6. ОГРАНИЧЕНИЯ ДЛЯ ЗАПРОСОВ К БАЗАМ ДАННЫХ

Запросы к базам данных — самые затратные и продолжительные вычисления во многих сервисах, включая этот. По многим причинам, включая перечисленные ниже, пакетный сервис ETL должен ограничиваться по частоте и допустимой продолжительности выполнения запросов:

- Многие сервисы баз данных являются общими. Каждый пользователь, запускающий длинные и затратные запросы, значительно снижает остаточную

емкость сервиса для обслуживания запросов других пользователей и увеличивает задержку. Запросы потребляют ресурсы процессора и памяти на своих хостах. Каждое подключение к сервису базы данных также потребляет программный поток; процесс этого потока выполняет запрос, собирает и возвращает результаты этого запроса. Можно создать пул с ограниченным количеством потоков, чтобы в приложении никогда не возникало слишком большого количества параллельных запросов.

- Ваши сервисы баз данных могут поддерживаться сторонними облачными провайдерами, размер платы за которые зависит от интенсивности использования, и затратные долго выполняемые запросы будут обходиться дорого.
- Сервис пакетных заданий ETL использует планирование выполнения запросов. Каждый запрос должен быть выполнен за отведенное для него время. Например, ежечасный запрос должен выполняться не более часа.

Можно реализовать средства для парсинга определений пользовательских запросов, когда пользователь создает их в конфигурации задания или отправляет запрос вместе с остальной конфигурацией задания на бэкенд.

В этом разделе будут рассмотрены возможные ограничения для пользовательских запросов, чтобы система соответствовала требованиям и допустимым затратам.

10.6.1. Ограничение времени выполнения запросов

Простой способ не допустить затратных запросов основан на ограничении времени выполнения запросов 10 минутами, когда владелец создает или редактирует конфигурацию задания, и 15 минутами при выполнении задания. Когда пользователь создает или редактирует запрос в конфигурации задания, перед сохранением строки запроса бэкенд должен потребовать, чтобы пользователь запустил запрос, и проверить, что его выполнение заняло менее 10 минут. Таким образом пользователи привыкнут соблюдать 10-минутное ограничение по запросам. Альтернатива — неблокирующий/асинхронный режим. Разрешите пользователю сохранить запрос, выполните его, а затем оповестите пользователя по электронной почте или в чате, был ли его запрос успешно выполнен за 10 минут и, соответственно, принята или отклонена конфигурация задания. С другой стороны, владельцы не готовы лишний раз изменять строки запросов; из-за этого некоторые ошибки могут остаться неисправленными, а улучшения — невнесенными.

Возможно, стоит запретить нескольким пользователям одновременно редактировать запрос и перезаписывать обновления других пользователей. О том, как предотвратить подобные конфликты, рассказано в разделе 2.4.2.

Если выполнение запроса занимает более 15 минут, завершите запрос, заблокируйте задание до тех пор, пока владелец не отредактирует и не проверит запрос,

и отправьте неотложное оповещение владельцам. Если время выполнения запроса превышает 10 минут, инициируйте некритичное оповещение для владельцев конфигурации задания, предупреждающее, что в будущем их запрос может превысить ограничение в 15 минут.

10.6.2. Проверка строк запросов перед отправкой

Заставлять пользователя по несколько минут ожидать сохранения конфигурации задания или сообщать о том, что операция была отклонена, лишь через 10 минут после сохранения конфигурации — не лучший вариант. Гораздо удобнее, если пользовательский интерфейс будет предоставлять обратную связь по строкам запросов сразу в процессе их создания, чтобы пользователь не отправил конфигурацию задания с недействительными или затратными запросами. Такая проверка может включать следующие параметры.

Запрет полного сканирования таблиц. Разрешайте только запросы к таблицам, содержащим ключи секций; при этом запросы должны содержать фильтры по ключам секций. Можно пойти еще дальше и ограничить количество ключей секций в запросе. Чтобы определить ключи секций таблицы, бэкенд должен выполнить запрос DESCRIBE к соответствующему сервису базы данных. Не разрешайте запросы, содержащие операцию JOIN, — они могут оказаться чересчур затратными.

После того как пользователь задаст запрос, можно вывести план выполнения, чтобы пользователь мог настроить запрос на минимизацию времени его выполнения. Эта функция должна сопровождаться ссылками на руководства по настройке запросов на соответствующем языке базы данных. За информацией о настройке запроса SQL обращайтесь к https://www.toptal.com/sql-server/sql-database-tuning-for-developers. За информацией о настройке запросов Hive обращайтесь к https://cwiki.apache.org/conluence/display/Hive/LanguageManual+Explain или главе Performance Considerations книги Дайонг Ду (Dayang Du) «Apache Hive Essentials», Packt Publishing, 2018.

10.6.3. Раннее обучение пользователей

Пользователи, создающие запросы, должны узнавать о существующих ограничениях в начале работы, чтобы приспособиться к ним. Также следует предоставить хороший интерфейс и содержательную документацию, из которой пользователи могут узнать об ограничениях. Более того, ограничения желательно определять и задавать в ранней версии сервиса пакетного аудита базы данных, а не добавлять их через несколько месяцев после первого выпуска. Если пользователям было разрешено отправлять затратные запросы до введения ограничений, они могут начать возражать против них, и убедить их изменить запросы будет трудно или невозможно.

10.7. КАК ИЗБЕЖАТЬ МНОЖЕСТВА ОДНОВРЕМЕННЫХ ЗАПРОСОВ

Следует установить лимит на количество запросов, которые могут выполняться пакетным сервисом ETL одновременно. Каждый раз, когда пользователь передает конфигурацию задания, которая будет содержать запрос для выполнения по определенному плану, бэкенд должен проверить количество запросов, запланированных для одновременного выполнения с одной базой данных, и отправить оповещение разработчикам сервиса, если количество одновременных запросов приблизится к пределу пропускной способности. Можно отслеживать время ожидания каждого запроса перед началом его выполнения и выдавать некритичные оповещения, если время ожидания превышает 30 минут или другое выбранное эталонное значение. Также можно подумать о проектировании схем нагрузочного тестирования для оценки пропускной способности. Обновленная высокоуровневая архитектура представлена на рис. 10.4.

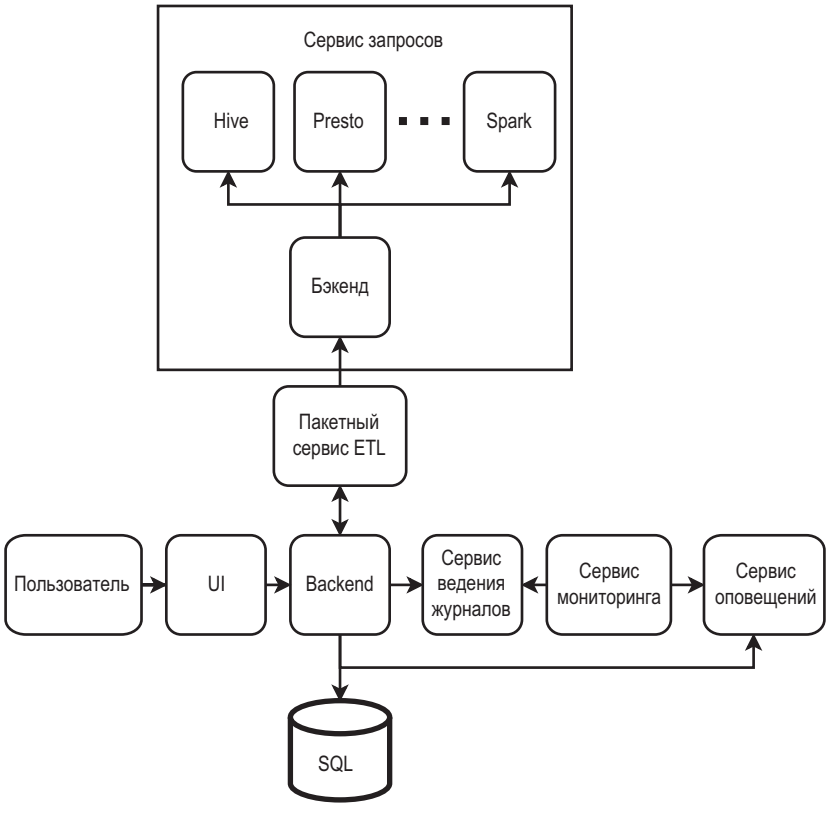

Рис. 10.4. Обновленная высокоуровневая архитектура с общим сервисом запросов, через который другие сервисы отправляют запросы к базам данных

На рис. 10.4 представлен новый сервис запросов баз данных. Так как базы данных представляют собой общие сервисы, сквозная функциональность (такая, как настроенное ограничение количества одновременных запросов) должна храниться в сервисе запросов базы данных, а не в сервисе аудита базы данных.

Другой способ оптимизации заключается в том, что сервис пакетных заданий ETL перед выполнением запроса к базе данных может обратиться к сервису уведомлений через сервис бэкенда, чтобы проверить, не осталось ли других необработанных оповещений. В таком случае обрабатывать задание аудита не обязательно.

10.8. ДРУГИЕ ПОЛЬЗОВАТЕЛИ МЕТАДАННЫХ СХЕМЫ БАЗЫ ДАННЫХ

Чтобы помочь пользователям в создании запросов, сервис может автоматически определять конфигурации заданий по метаданным схемы. Например, фильтры `WHERE` обычно определяются для столбцов секций, чтобы пользовательский интерфейс мог представить шаблоны запросов, которые предлагают эти столбцы пользователю или же предлагают создать запрос, который проверяет только самую большую секцию.

По умолчанию, если новая секция проходит аудит, сервис не должен планировать другие аудиты для этой секции. У пользователей могут быть причины повторно выполнить тот же аудит, несмотря на то что он пройден. Например, задание аудита может содержать ошибки, его прохождение будет некорректным, а владельцу задания придется отредактировать задание аудита и повторно выполнить прошедшие аудиты. Следовательно, сервис должен предоставить пользователю возможность вручную повторить аудит или запланировать ограниченное количество аудитов для этой секции.

Для таблиц может быть определено соглашение SLA о частоте присоединения новых ролей. Это относится к концепции *свежести данных*, то есть их актуальности. Аудит таблицы не должен выполняться, пока данные не будут готовы, так как это будет неэффективно и приведет к ложным оповещениям. Возможно, сервис запросов баз данных может реализовать механизм, позволяющий владельцам таблиц настраивать SLA свежести данных для своих таблиц, или же разработать каталог/платформу метаданных баз данных организации с использованием таких инструментов, как Amundsen (https://www.amundsen.io/), DataHub (https://datahubproject.io/) или Metacat (https://github.com/Netlix/metacat).

Другая полезная возможность платформы метаданных баз данных — запись инцидентов, относящихся к их таблицам. Владелец таблицы или вашего сервиса может обновить платформу метаданных баз данных информацией о том, что у какой-то таблицы возникли проблемы. Сервис запросов баз данных может

предупредить любого человека или сервис, обращающийся с запросами к этой таблице, об ошибках аудита. Пользователь, обратившийся с запросом к таблице, может снова обратиться к ней в будущем, так что в платформе метаданных баз данных полезно реализовать возможность подписки на изменения в метаданных таблицы или оповещения о проблемах, относящихся к таблице.

Сервис пакетных заданий ETL также может отслеживать изменения в схеме базы данных и реагировать соответствующим образом. Если имя столбца изменяется. Если имя столбца изменяется, сервис должен обновить его в строках запроса конфигураций заданий аудита, которые содержат это имя. Если столбец удаляется, это должно привести к блокировке всех заданий, в которых он используется, и оповещению их владельцев.

10.9. АУДИТ ПАЙПЛАЙНА ДАННЫХ

На рис. 10.5 изображен пайплайн данных (такой, как направленный ациклический граф Airflow) с несколькими заданиями. Каждое задание записывает данные в определенные таблицы, которые читаются на следующем уровне. Конфигурация задания может содержать поля для имени пайплайна и уровня, которые могут быть представлены дополнительными столбцами таблицы `job_config`.

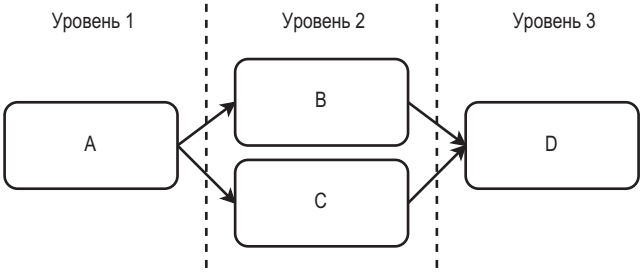

Рис. 10.5. Пайплайн данных с несколькими уровнями. Для каждого уровня можно создать задание аудита

Если конкретное задание аудита завершается неудачей, сервис должен выполнить следующие действия:

- Заблокировать нижележащие аудиты (то есть аудиты следующих уровней) для экономии ресурсов, так как выполнение заданий аудита в случае сбоя предшествующего задания станет бессмысленной тратой ресурсов.
- Заблокировать другие задания, содержащие запросы к этой таблице, и их нисходящие задания.
- Отправить срочные оповещения владельцам всех заблокированных заданий и владельцам всех нисходящих заданий.

Также следует обновить платформу метаданных баз данных информацией о проблемах с таблицей. Любой пайплайн данных, использующих эту таблицу, должен заблокировать все задачи, нижележащие по отношению к этой таблице, иначе на них могут распространиться некорректные данные из этой таблицы. Например, пайплайны машинного обучения могут определять, должны ли они выполняться, на основании результатов аудита, чтобы эксперименты не проводились с некорректными данными. Airflow позволяет пользователям настраивать правила срабатывания (https://airlow.apache.org/docs/apache-airlow/stable/concepts/dags.html#trigger-rules), чтобы каждая задача выполнялась только в том случае, если все ее зависимости (или хотя бы одна) успешно завершили выполнение. Функциональность сервиса пакетных заданий ETL хорошо дополняет Airflow и другие платформы управления рабочими процессами.

Все сказанное наводит на мысль о том, что сервис пакетных заданий ETL можно преобразовать в общий сервис, чтобы эта функциональность объединения заданий ETL в пакеты поддерживалась в масштабе организации.

Когда пользователь добавляет в пайплайн новый уровень, он также должен обновить значения уровней всех нисходящих задач. Как показано на рис. 10.6, бэкенд может помочь ему, автоматически повышая уровни всех нижележащих задач.

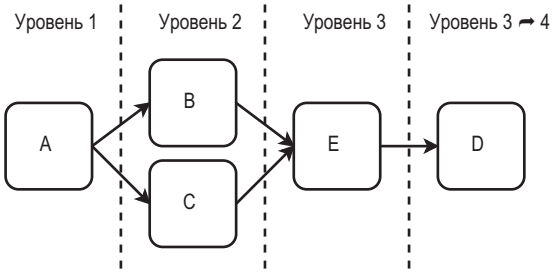

Рис. 10.6. При добавлении новой задачи E между уровнями 2 и 3 автоматически увеличивается номер соответствующего уровня, так что уровень 3 становится уровнем 4

10.10. ВЕДЕНИЕ ЖУРНАЛОВ, МОНИТОРИНГ И ОПОВЕЩЕНИЯ

В дополнение к тому, что обсуждалось в разделе 2.5, следует организовать мониторинг и отправку оповещений по некоторым дополнительным условиям, перечисленным ниже. Следующие данные журналов могут быть полезны пользователям и отображаться в пользовательском интерфейсе:

- Текущий статус задания (запущено, выполняется, завершилось успешно, завершилось с ошибкой) и время регистрации этого статуса.

- Неудачные запросы к базе данных сервиса пакетных заданий ETL. Оповещение также должно содержать причины сбоя: например, тайм-аут запроса или ошибка при выполнении запроса.
- Как уже говорилось, необходимо отслеживать время, затраченное на выполнение запросов к базам данных, и выдавать оповещения при превышении установленного эталонного значения.
- Как уже отмечалось, следует оповещать владельцев заданий о сбоях вышележащих заданий.
- Значение 1 секунда для 99-го процентиля и ответы 4xx и 5xx конечных точек бэкенда.
- Значение 1 секунда для 99-го процентиля и ответы 4xx и 5xx по запросам к внешним сервисам.
- Высокий трафик, определяемый превышением частоты запросов предельной нагрузки, установленной на основе нагрузочного тестирования.
- Высокая нагрузка на процессор, память или систему ввода/вывода.
- Высокие затраты памяти в сервисе SQL (если вы управляете собственным сервисом SQL а не общим).

Ответы 4xx должны инициировать выдачу срочных оповещений, тогда как для других проблем можно выдавать оповещения с низкой критичностью.

10.11. ДРУГИЕ ВОЗМОЖНЫЕ ТИПЫ АУДИТА

Кроме рассмотренных операций аудита/проверок, можно обсудить другие разновидности тестов. Назовем некоторые из них.

10.11.1. Аудит согласованности данных между датацентрами

Одни и те же данные нередко хранятся в нескольких датацентрах. Чтобы обеспечить согласованность данных между датацентрами, сервис пакетного аудита баз данных может предоставить возможность проведения выборочных тестов для сравнения данных между датацентрами.

10.11.2. Сравнение выше- и нижележащих данных

Вспомните раздел 7.7, о миграции данных. Пользователю может понадобиться скопировать данные из одной таблицы в другую. Он может создать задание аудита для сравнения новейших секций вышележащих и нижележащих таблиц, чтобы проверить согласованность данных.

10.12. ДРУГИЕ ВОЗМОЖНЫЕ ТЕМЫ ОБСУЖДЕНИЯ

Другие темы, которые также могут обсуждаться в ходе собеседования:

- Проектирование масштабируемого сервиса пакетных заданий ETL или масштабируемого сервиса оповещений. Для обоих сервисов понадобится распределенная платформа потоковой передачи событий (например, Kafka).
- Программирование функции, генерирующей задание Python для Airflow на базе validation.py.template и других подходящих шаблонов, с отдельной задачей Airflow для каждого запроса. Впрочем, это вопрос из области программирования, а не проектирования систем.
- Задание аудита оповещает владельцев таблиц баз данных о проблемах с целостностью данных в их таблицах, но мы не обсуждали, как диагностировать и выявлять причины этих проблем. Как владельцу таблицы диагностировать проблемы целостности данных? Можно ли усовершенствовать сервис аудита? Какие еще существуют возможности оптимизации?
- Некоторые задания аудита могут завершиться с ошибкой при одном запуске, но быть успешно выполнены, когда владелец запустит тот же запрос в процессе диагностики. Как владельцу провести диагностику таких заданий и какие средства или данные журналов может предоставить сервис для упрощения этой задачи?
- Как обнаружить и провести дедупликацию идентичных или очень похожих заданий аудита?
- Сервис пакетного аудита баз данных отправляет большое количество оповещений. Проблема с таблицей может затронуть несколько заданий аудита и привести к выдаче нескольких оповещений для одного пользователя. Как удалить дубликаты оповещений? Какие части логики дедупликации оповещений будут реализованы в сервисе пакетного аудита баз данных, а какие — в общем сервисе оповещений?
- Сервис также может поддерживать запуск тестов по определенному событию в отличие от простого запуска по плану. Например, можно отслеживать количество строк, изменяющихся после каждого запроса, суммировать эти значения и запускать тест после изменения заданного количества строк данных. Можно обсудить, какие события могут инициировать запуск тестов и их место в дизайне системы.

10.13. ССЫЛКИ

Текст этой главы был написан на основе материалов платформы качества данных Uber Trust, однако многие подробности реализации, которые мы рассмотрели, могут существенно отличаться от Trust. Обсуждение качества данных в Uber

доступно на странице https://eng.uber.com/operational-excellence-data-quality/, хотя в этой статье название Trust не упоминается. В статье приведен обзор платформы качества данных Uber, включая обсуждение входящих в нее сервисов и их взаимодействий друг с другом и с пользователем.

ИТОГИ

- В ходе собеседования по проектированию систем можно обсудить аудит как стандартный подход к обеспечению целостности данных. В этой главе рассматривается возможный системный дизайн пакетного аудита.
- Можно периодически запускать запросы к базам данных для обнаружения аномалий, которые могут быть обусловлены разными проблемами: непредвиденной активностью пользователей, скрытыми ошибками или вредоносной деятельностью.
- Мы разработали популярное решение для выявления аномалий данных, которое охватывает многие практические сценарии периодических запросов к базам данных, и спроектировали масштабную, доступную и точную систему.
- Платформы планирования задач (такие, как Airflow) могут использоваться для планирования заданий аудита — вместо заданий cron, которые хуже масштабируются и менее надежны.
- Необходимо предусмотреть средства мониторинга и оповещения, чтобы пользователи знали об успешном или неудачном выполнении заданий аудита. Сервис периодического аудита баз данных также использует сервис оповещений, рассмотренный в главе 9, и OpenID Connect, рассматриваемый в приложении Б.
- Можно предоставить сервис запросов, при помощи которого пользователи смогут создавать ситуационные запросы.

11 Автозаполнение/ опережающий ввод

В ЭТОЙ ГЛАВЕ

- ✓ Сравнение автозаполнения с поиском
- ✓ Отделение сбора и обработки данных от получения информации
- ✓ Обработка непрерывного потока данных
- ✓ Разделение большого агрегирующего пайплайна на стадии для сокращения затрат на хранение данных
- ✓ Использование побочных результатов пайплайнов обработки данных для других целей

Требуется спроектировать систему автозаполнения. Это полезное задание для проверки способности кандидата спроектировать распределенную систему, которая непрерывно поглощает и обрабатывает большие объемы данных и превращает их в компактную (несколько мегабайт) структуру данных. К этой структуре пользователи могут обращаться с запросами для конкретной цели. Система автозаполнения получает данные из строк, отправляемых миллиардами пользователей, а затем перерабатывает эти данные во взвешенное префиксное дерево (trie). Когда пользователь вводит строку, взвешенное префиксное дерево предоставляет ему рекомендации по автозаполнению. Также в систему автозаполнения можно добавить персонализацию и элементы машинного обучения.

11.1 ВОЗМОЖНОЕ ПРИМЕНЕНИЕ АВТОЗАПОЛНЕНИЯ

Сначала стоит обсудить и уточнить предполагаемые практические сценарии использования этой системы, чтобы вы были уверены в том, что верно определили требования. Возможные применения автозаполнения:

- Дополнение к сервису поиска. Когда пользователь вводит поисковый запрос, сервис автозаполнения возвращает список рекомендаций автозаполнения при каждом нажатии клавиши. Если пользователь выбирает предложенный вариант, поисковый сервис принимает его и возвращает список результатов.
- Общие поисковые системы (Google, Bing, Baidu, Yandex и т. д.).
- Поиск по определенной коллекции документов. Примеры: Википедия и приложения видеохостинга.
- Текстовые редакторы могут предоставлять рекомендации по автозаполнению. Когда пользователь начинает вводить слово, редактор может предоставить рекомендации по автозаполнению для часто встречающихся слов, начинающихся с введенного префикса. Методом *нечеткого сопоставления* функциональность автозаполнения также может использоваться для проверки орфографии; она предлагает слова с префиксами, близкими к введенному префиксу, но не совпадающему с ним полностью.
- Интегрированная среда разработки (IDE) для написания кода может содержать функцию автозаполнения. Функция автозаполнения может отслеживать имена переменных или констант из каталога проекта и предоставлять их в виде рекомендаций автозаполнения, когда пользователь объявляет переменную или константу. При этом необходимы точные совпадения (нечеткое сопоставление не подойдет).

Сервис автозаполнения во всех этих случаях будет иметь разные источники данных и архитектуры. Потенциальная ошибка на собеседовании — сделать поспешные выводы и предположить, что сервис автозаполнения предназначен для сервиса поиска; ее часто совершают те, кто работает с автозаполнением в таких поисковых системах, как Google или Bing.

Даже если эксперт дает конкретное задание, например «Спроектируйте систему, которая предоставляет функциональность автозаполнения для общих поисковых приложений, таких как Google», уделите полминуты на обсуждение других возможностей применения автозаполнения. Покажите, что вы не ограничиваетесь рамками задачи и не делаете поспешных предположений или выводов.

11.2. ПОИСК И АВТОЗАПОЛНЕНИЕ

Следует разделять автозаполнение и поиск, а также их требования. Тогда у вас получится спроектировать именно сервис автозаполнения, а не поиска. Чем

автозаполнение похоже на поиск и чем отличается от него? Среди похожих свойств можно выделить следующие:

- Обе разновидности сервисов определяют намерения пользователя на основании введенной им строки и выводят список результатов, отсортированный по вероятности соответствия.
- Чтобы избежать возврата для пользователя неподходящего контента, обе разновидности сервисов должны провести предварительную обработку возможных результатов.
- Оба сервиса могут регистрировать пользовательский ввод и использовать его для улучшения рекомендаций/результатов. Например, оба сервиса могут регистрировать возвращенные результаты и вариант, который выбрал пользователь. Если он выберет первый вариант, это означает, что такой вариант более релевантен.

Автозаполнение концептуально проще поиска. Некоторые высокоуровневые различия описаны в табл. 11.1. Если только эксперт не выразит особого интереса, не тратьте больше минуты на их обсуждение на собеседовании. Главное, продемонстрировать критическое мышление и умение видеть общую картину.

Таблица 11.1. Некоторые различия между поиском и автозаполнением

Поиск	Автозаполнение
Результаты обычно представляют собой список URL-адресов веб-страниц или документов. Эти документы проходят предварительную обработку для создания индекса. При выполнении поискового запроса строка сопоставляется с индексом для получения соответствующих документов	Результаты представляют собой списки строк, сгенерированные на основании поисковых строк других пользователей
Задержка 99-го процентиля продолжительностью в несколько секунд может быть приемлемой. Более высокая задержка (до минуты) может быть приемлемой в некоторых случаях	Для оптимального взаимодействия с пользователем желательна низкая задержка 99-го процентиля приблизительно 100 мс. Пользователи ожидают, что рекомендации будут появляться практически сразу после ввода каждого символа
Возможны разные типы данных результатов, включая строки, составные объекты, файлы и мультимедиа	Типом данных результата всегда является обычная строка
Каждому результату назначается метрика релевантности	Не всегда характеризуется релевантностью. Например, список результатов завершения в IDE может быть упорядочен лексикографически

Таблица 11.1 (*окончание*)

Поиск	Автозаполнение
Высокие требования к точности вычисления метрики релевантности (где точность воспринимается пользователем субъективно)	Требования к точности (например, пользователь выбирает один из первых предложенных вариантов, а не последних) могут быть не такими строгими, как при поиске. Это сильно зависит от бизнес-требований, и иногда высокая точность может быть необходимой
Результат поиска может вернуть любые из входящих документов. Это означает, что документ должен быть обработан, проиндексирован и, возможно, возвращен в результатах поиска. Для снижения сложности можно сделать выборку из содержимого документа, но обработать необходимо каждый отдельный документ	Если высокая точность не обязательна, для снижения сложности могут применяться такие методы, как выборка и приближенные алгоритмы
Может возвращать сотни результатов	Обычно возвращает 5–10 результатов
Пользователь может выбрать несколько результатов, щелкнув на кнопке возврата и выбрав другой результат. Это механизм обратной связи, из которого можно извлечь много полезной информации	Другой механизм обратной связи. Если ни одна из рекомендаций автозаполнения не подходит, пользователь завершает ввод строки и отправляет ее для поиска

11.3. ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

Можно обсудить следующие вопросы для проработки функциональных требований системы автозаполнения.

11.3.1. Область применения сервиса автозаполнения

Сначала можно прояснить некоторые подробности области применения, например, какие практические сценарии и языки должны поддерживаться:

- Предназначена ли система автозаполнения для общего сервиса поиска или для других инструментов, например текстовых редакторов или IDE?
  - Система предназначена для предложения поисковых строк в общем сервисе поиска.
- Предназначена ли она только для английского языка?
  - Да.

- Сколько слов она должна поддерживать?
  - Словарь английского языка Webster содержит около 470 000 слов (https://www.merriam-webster.com/help/faq-how-many-english-words), тогда как Оксфордский словарь английского языка — более 171 000 слов (https://www.lexico.com/explore/how-many-words-are-there-in-the-english-language). Мы не знаем, сколько из этих слов содержат не менее 6 символов, поэтому не будем делать предположений. Возможно, вы захотите поддерживать популярные слова, еще не включенные в словарь, поэтому будем поддерживать набор до 100 000 слов. В английском языке средняя длина слова равна 4,7 символа (округляем до 5), по 1 байту на букву — требования к памяти составляют всего 5 Мбайт. Возможность ручного (но не программного) добавления слов и фраз увеличивает объем памяти, но это увеличение будет пренебрежимо малым.

ПРИМЕЧАНИЕ IBM 350 RAMAC, появившийся в 1956 году, был первым компьютером с жестким диском на 5 Мбайт (https://www.ibm.com/ibm/history/exhibits/650/650_pr2.html). Он весил больше тонны и занимал площадь 9 × 15 м^2. Программирование выполнялось на машинном языке, с проводных перемычек на штекерной панели. В те дни собеседования по проектированию систем еще не проводились.

11.3.2. Особенности UX

Можно уточнить некоторые особенности UX (User Experience, пользовательский опыт) в отношении рекомендаций автозаполнения, например, должны ли предлагаемые варианты иметь форму предложений или отдельных слов и сколько символов должен ввести пользователь до появления рекомендаций:

- Применяется ли автозаполнение к словам или предложениям?
  - Начнем со слов, а затем расширим автозаполнение до отдельных выражений или предложений, если будет время.
- Существует ли минимальное количество символов, которое должно быть введено перед предоставлением вариантов рекомендаций?
  - 3 символа — оптимальный вариант.
- Существует ли минимальная длина слов, для которых будут предоставляться рекомендации? Получать рекомендации для слов из 4 или 5 букв после ввода 3 символов неэффективно, так как это сэкономит всего 1 или 2 буквы.
  - Будем рассматривать слова, содержащие не менее 6 букв.
- Следует ли учитывать цифры и специальные символы или только буквы?
  - Только буквы. Цифры и специальные символы игнорируются.

- Сколько рекомендаций автозаполнения выводить за один раз и в каком порядке?
  - Будем выводить по 10 рекомендаций, упорядоченных по убыванию частоты. Начнем с предоставления конечной точки API GET, которая получает строку и возвращает список из 10 слов из словаря, упорядоченных по убыванию приоритета. Затем решение можно будет расширить, чтобы оно получало идентификатор пользователя для возвращения персонализированных рекомендаций.

11.3.3. История поиска

Необходимо решить, должны ли рекомендации автозаполнения быть основаны только на текущем вводе пользователя или же на его истории поиска и других источниках данных.

- Ограничение рекомендаций набором слов означает, что придется обрабатывать строки, отправленные пользователями. Если результатом этой обработки становится индекс, на основе которого выводятся рекомендации автозаполнения, должны ли ранее обработанные данные обрабатываться повторно, чтобы включать такие вручную добавленные или удаленные слова/выражения?
  - Подобные вопросы свидетельствуют об инженерном опыте. Следует обсудить с экспертом, что объем имеющихся данных, которые нуждаются в повторной обработке, будет весьма значителен. Но зачем добавлять новое слово или выражение вручную? Процедура должна базироваться на аналитике прошлых поисковых строк пользователя. Стоит рассмотреть пайплайн ETL, где легко создать таблицы, к которым можно обращаться с запросами, чтобы получить аналитику и другую полезную информацию.
- Что выбрать в качестве источника данных для рекомендаций? Только ранее отправленные запросы или также другие источники данных (например, демографические данные пользователей)?
  - Желательно рассмотреть другие источники данных. Пока мы ограничимся отправленными запросами. Возможно, в будущем стоит подумать о расширяемом дизайне с добавлением других источников данных.
- Основывать ли рекомендации на всех пользовательских данных или только на данных текущего пользователя (то есть персонализированное автозаполнение)?
  - Начнем со всех пользовательских данных, а затем рассмотрим персонализацию.
- Какой период времени использовать для рекомендаций?
  - Сначала рассмотрим все время; затем можно удалить данные старше одного года. Можно воспользоваться пороговой датой — скажем, не рассматривать данные до 1 января прошлого года.

11.3.4. Модерирование и качество контента

Можно рассмотреть и другие средства, такие как модерирование контента и его качество:

- Как насчет механизма, который позволяет пользователям сообщать о неподходящих рекомендациях?
  - Это полезная функция, но пока мы не будем ею заниматься.
- Нужно ли учитывать, что небольшая группа пользователей может отправлять большую часть поисковых запросов? Должен ли сервис автозаполнения учитывать интересы большинства пользователей, обрабатывая одинаковое количество поисковых запросов на пользователя?
- Нет, будем рассматривать только сами поисковые строки, не учитывая, какие пользователи их отправили.

11.4. НЕФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

После обсуждения функциональных требований можно таким же образом уточнить и нефункциональные. При этом можно затронуть возможные компромиссы, например доступность/производительность:

- Система должна быть масштабируемой, чтобы к ней могли обращаться глобально все пользователи.
- Высокая доступность не обязательна. Это не критичный параметр, так что отказоустойчивостью можно в определенной степени пожертвовать.
- Высокая производительность и пропускная способность необходимы. Пользователь должен увидеть рекомендации автозаполнения через полсекунды.
- Консистентность необязательна. Допустимо, чтобы рекомендации отставали на несколько часов; рекомендации не обязательно немедленно обновлять новыми поисковыми запросами.
- Что касается конфиденциальности и безопасности, автозаполнение не требует авторизации или аутентификации, но данные пользователей должны быть защищены.
- В отношении точности можно руководствоваться следующими соображениями:
  - Рекомендации могут возвращаться с учетом частоты поиска, поэтому можно добавить счетчики поисковых строк. Будем исходить из того, что такой счетчик не обязан быть точным, и на первой итерации проектирования может быть достаточно приближенного значения. Повышением точности, в том числе определением метрик точности, можно заняться при наличии времени.

- Мы не будем рассматривать опечатки или запросы, содержащие текст на разных языках. Проверка орфографии полезна, но здесь мы это опустим.
- Что касается потенциально недопустимых слов и выражений, можно ограничить рекомендации набором, исключающим недопустимые слова, но не выражения. Будем называть их словарными словами, хотя к ним могут относиться и слова, добавленные вручную, а не взятые из словаря. При желании можно спроектировать механизм, позволяющий администраторам вручную добавлять и удалять слова и выражения из этого набора.
- Для актуальности рекомендаций можно установить неформальное требование в 1 день.

11.5. ПЛАНИРОВАНИЕ ВЫСОКОУРОВНЕВОЙ АРХИТЕКТУРЫ

На собеседовании по проектированию систем можно начать с эскиза исходной архитектуры очень высокого уровня, такого как на рис. 11.1. Пользователи отправляют поисковые запросы, которые обрабатываются системой поглощения данных и сохраняются в БД. Пользователи получают рекомендации автозаполнения из базы данных во время ввода поисковых строк. До этапа получения рекомендаций автозаполнения могут присутствовать промежуточные шаги; назовем их системой запросов. Эта диаграмма поможет представить общую картину сервиса.

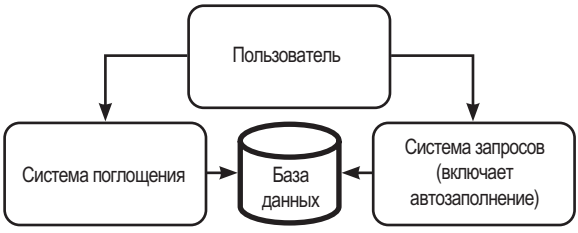

Рис. 11.1. Высокоуровневая исходная архитектура сервиса автозаполнения. Пользователи отправляют строки поисковых запросов системе поглощения данных, и эти строки сохраняются в базе данных. Пользователи отправляют запросы к системе запросов для получения рекомендаций автозаполнения. Мы еще не обсуждали, где происходит обработка данных

Следующий логичный шаг — разделение системы на такие компоненты:

1. Поглощение данных.
2. Обработка данных.

3. Запрос к обработанным данным для получения рекомендаций автозаполнения.

Обработка данных обычно требует больше ресурсов, чем поглощение. Поглощение должно только принимать и регистрировать запросы, а также обрабатывать выбросы трафика. Таким образом, чтобы провести масштабирование, мы отделяем систему обработки данных от системы поглощения. Это пример паттерна проектирования «Разделение ответственности на команды и запросы» (CQRS), обсуждавшийся в главе 1.

Другой фактор, который необходимо учитывать: система поглощения может быть сервисом ведения журналов сервиса поиска, который в свою очередь может быть общим сервисом ведения журналов организации.

11.6. ВЗВЕШЕННЫЕ ПРЕФИКСНЫЕ ДЕРЕВЬЯ И ИСХОДНАЯ ВЫСОКОУРОВНЕВАЯ АРХИТЕКТУРА

На рис. 11.2 изображена исходная высокоуровневая архитектура системы автозаполнения. Система автозаполнения не является отдельным сервисом; это система, в которой пользователи только обращаются с запросом к одному

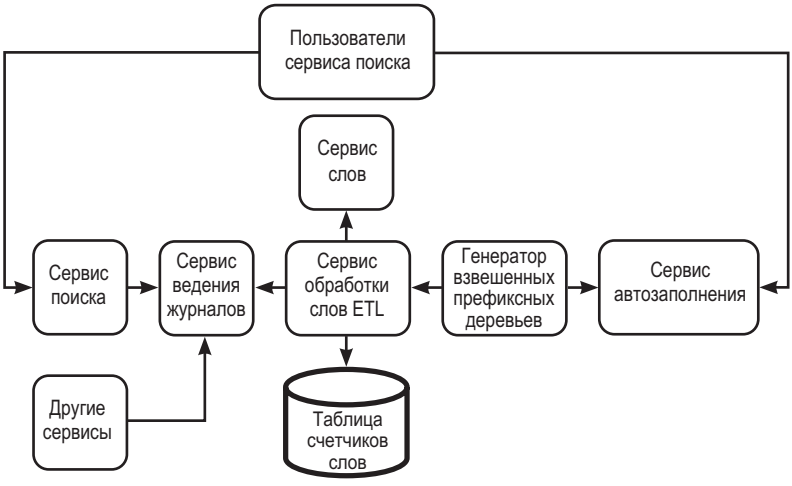

Рис. 11.2. Исходная высокоуровневая архитектура сервиса автозаполнения. Пользователи сервиса поиска отправляют поисковые строки сервису поиска, и эти строки регистрируются общим сервисом ведения журналов. Задание обработки слов ETL может быть пакетным или потоковым, читающим и обрабатывающим сохраненные поисковые строки. Генератор взвешенных префиксных деревьев читает счетчик слов, генерирует взвешенное префиксное дерево, а затем отправляет его сервису автозаполнения, от которого пользователи получают рекомендации

сервису (сервису автозаполнения) и не взаимодействуют напрямую с остальными компонентами системы. Остальные компоненты предназначены для того, чтобы собирать пользовательские поисковые строки и периодически генерировать и отправлять взвешенное префиксное дерево в сервис автозаполнения.

Общий сервис ведения журналов — низкоуровневый источник данных, на основании которого сервис автозаполнения формирует рекомендации, которые он предоставляет пользователям. Пользователи сервиса поиска отправляют свои запросы сервису поиска, который сохраняет их в сервис ведения журналов. Другие сервисы также сохраняют свои запросы в общий сервис ведения журналов. Сервис автозаполнения может обратиться с запросом к сервису ведения журналов для получения журналов сервиса поиска или других сервисов, если вы решите, что это будет полезно для улучшения рекомендаций автозаполнения.

Общий сервис ведения журналов должен иметь API для получения сообщений журналов в зависимости от темы и метки времени. Можно указать эксперту, что детали его реализации (например, используемая база данных — MySQL, HDFS, Kafka, Logstash и т. д.) не актуальны для текущего обсуждения, так как вы проектируете сервис автозаполнения, а не общий сервис ведения журналов организации. Добавьте, что вы готовы обсудить подробности реализации общего сервиса ведения журналов, если понадобится.

Пользователи получают рекомендации автозаполнения от бэкенда сервиса автозаполнения. Рекомендации генерируются с использованием взвешенного префиксного дерева, изображенного на рис. 11.3. Когда пользователь вводит строку, она сопоставляется со взвешенным префиксным деревом. Список результатов генерируется из дочерних узлов совпавшей строки и сортируется по убыванию весов. Например, строка поиска «ba» возвращает результат `["bay", "bat"]`. `"bay"` имеет вес 4, тогда как `"bat"` имеет вес 2, поэтому `"bay"` будет предшествовать `"bat"`.

Обсудим подробную реализацию этих шагов.

11.7. ПОДРОБНАЯ РЕАЛИЗАЦИЯ

Генератор взвешенных префиксных деревьев может быть реализован в виде ежедневного пакетного пайплайна ETL (или потокового пайплайна, если необходимы обновления в реальном времени). Пайплайн включает задание обработки слов ETL. На рис. 11.2 задание обработки слов ETL и генератор взвешенных префиксных деревьев являются отдельными стадиями пайплайна, потому что задание обработки слов ETL может оказаться полезным для других целей и сервисов, а разделение стадий позволяет реализовать, тестировать, обслуживать и масштабировать их независимо друг от друга.

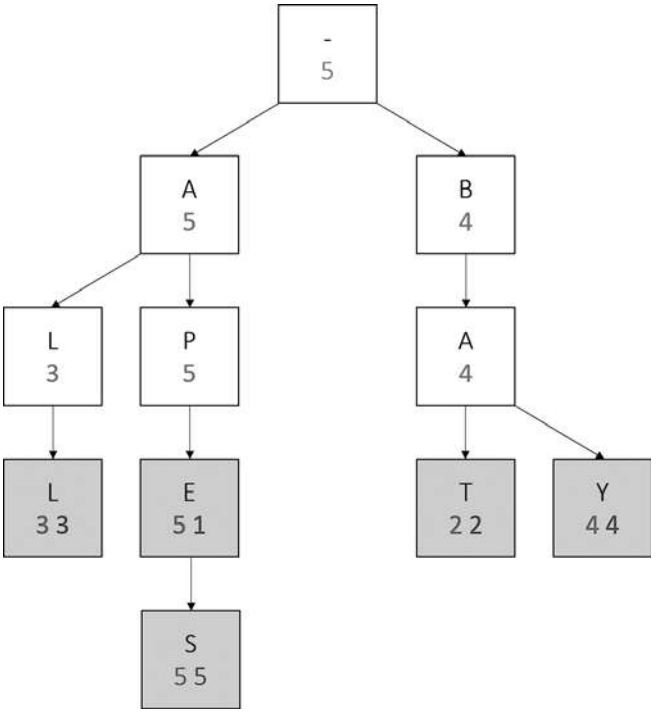

Рис. 11.3. Взвешенное префиксное дерево для слов «all», «apes», «bat» и «bay». (Источник: https://courses.cs.duke.edu/cps100/spring16/autocomplete/trie.html)

Пайплайн подсчета слов может состоять из следующих этапов, изображенных в виде направленного ациклического графа (DAG) на рис. 11.4:

1. Выборка соответствующих сохраненных данных из темы поиска сервиса ведения журналов (и возможно, других тем) и размещение их во временном хранилище.
2. Разбиение поисковых строк на слова.
3. Удаление недопустимых слов.
4. Подсчет слов и запись в таблицу счетчиков. В зависимости от требуемой точности можно подсчитывать каждое слово или воспользоваться приближенным алгоритмом, таким как Count-Min Sketch (раздел 17.7.1).
5. Фильтрация допустимых слов и сохранение частых неизвестных слов.
6. Генерация взвешенного префиксного дерева по таблице счетчиков слов.
7. Отправка взвешенного префиксного дерева хостам бэкенда.

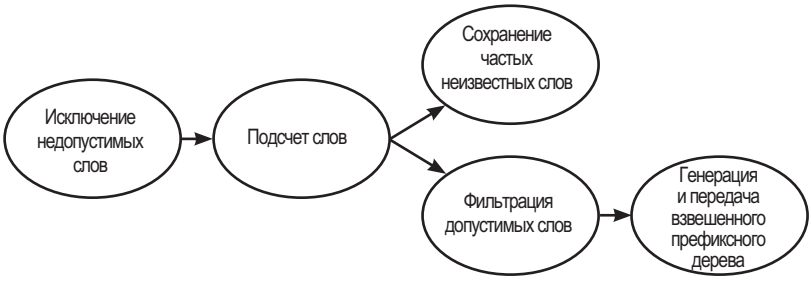

Рис. 11.4. DAG пайплайна подсчета слов. Запись частых неизвестных слов и фильтрация допустимых слов могут выполняться независимо

Для хранения журнала необработанных поисковых запросов можно рассмотреть разные технологии баз данных:

- Индекс Elasticsearch с секционированием по дням, часть типичного стека ELK с периодом удержания данных по умолчанию 7 дней.
- Журналы за каждый день могут быть представлены файлами HDFS (с секционированием по дням). Поисковые запросы пользователей могут отправляться в топик Kafka с периодом удержания, составляющим несколько дней (одного дня недостаточно на случай, если по какой-либо причине понадобится обработать старые сообщения). Ежедневно в назначенное время первая стадия пайплайна потребляет сообщения до тех пор, пока не обнаружит сообщение с меткой времени позднее заданной (это означает, что будет потреблено одно дополнительное сообщение, но эта небольшая неточность допустима) или топик не опустеет. Потребитель создает новый каталог HDFS для секции, соответствующей дате, и присоединяет все сообщения к одному файлу в этом каталоге. Каждое сообщение может содержать метку времени, идентификатор пользователя и поисковую строку. HDFS не предоставляет никаких механизмов настройки периода удержания, и для таких вариантов необходимо добавить в пайплайн стадию удаления старых данных.
- SQL не подойдет, поскольку требует размещения всех данных на одном узле.

Будем считать, что сервис ведения журналов является сервисом ELK. Как упоминалось в разделе 4.3.5, HDFS — типичное решение для хранения данных в модели программирования MapReduce. Мы будем использовать модель MapReduce для параллелизации обработки данных по нескольким узлам. С HDFS можно использовать Hive или Spark. При выборе Hive можно использовать Hive со Spark (https://spark.apache.org/docs/latest/sql-data-sources-hive-tables.html), так что в обоих вариантах — как с Hive, так и со Spark — фактически используется Spark. Spark позволяет читать и записывать данные из HDFS в память и обрабатывать данные в памяти, что намного быстрее, чем обработка на диске. В следующих разделах кратко обсуждаются реализации, использующие Elasticsearch, Hive

и Spark. На собеседованиях по проектированию систем код реализации подробно не рассматривается, достаточно краткого обсуждения.

В результате получаем стандартное задание ETL. Каждая стадия читает информацию из базы данных предыдущей стадии, обрабатывает данные и записывает их в базу данных, которая будет использоваться следующей стадией.

11.7.1. Каждая стадия должна быть независимой задачей

Вернемся к DAG пакетного сервиса ETL на рис. 11.4. Почему каждый шаг представлен независимой стадией? Когда вы только начинаете работать над минимально жизнеспособным продуктом, можно реализовать генерацию взвешенного префиксного дерева как отдельную задачу и просто объединить все функции в цепочку. Этот подход прост, но такую систему слишком сложно обслуживать. (Сложность и сопровождаемость взаимосвязаны, и простая система обычно проще в обслуживании, но в рассматриваемом примере корреляция обратная.)

Для отдельных функций можно реализовать тщательное модульное тестирование, чтобы свести к минимуму риск ошибок, реализовать ведение журналов для оставшихся ошибок, которые будут выявляться в рабочей среде, упаковывать все функции с ошибкой в блоки `try-catch` и сохранять эти ошибки в журнале. Однако некоторые ошибки могут остаться невыявленными, и если какая-то ошибка при генерировании взвешенного префиксного дерева приведет к сбою, весь процесс придется перезапустить с самого начала. Операции ETL создают интенсивную вычислительную нагрузку и могут выполняться несколько часов, так что подобное решение не отличается эффективностью. Следует реализовать эти шаги в виде отдельных задач и воспользоваться планировщиком (например, Airflow), чтобы каждая задача выполнялась только после успешного завершения предыдущей.

11.7.2. Выборка журналов из Elasticsearch в HDFS

Для Hive можно воспользоваться командой `CREATE EXTERNAL TABLE` (https://www.elastic.co/guide/en/elasticsearch/hadoop/current/hive.html#_reading_data_from_elasticsearch) и определить таблицу Hive для топика Elasticsearch. Затем записать журналы в HDFS командой Hive `INSERT OVERWRITE DIRECTORY '/path/to/output/dir' SELECT * FROM Log WHERE created_at = date_sub(current_date, 1);`. (Команда предполагает, что вам нужны вчерашние журналы.)

Для Spark можно воспользоваться методом SparkContext esRDD (https://www.elastic.co/guide/en/elasticsearch/hadoop/current/spark.html#spark-read) для подключения к топику Elasticsearch, за которым следует фильтрующий запрос Spark (https://spark.apache.org/docs/latest/api/sql/index.html#filter) для чтения данных за соответствующие даты, а затем записать информацию в HDFS с использованием

функции Spark `saveAsTextFile` (https://spark.apache.org/docs/latest/api/scala/org/apache/spark/api/java/JavaRDD.html#saveAsTextFile(path:String):Unit).

Во время собеседования, даже если вы не знаете, что в Hive и Spark существуют интеграции Elasticsearch, можно сказать эксперту, что такие интеграции могут существовать, потому что это популярные, общепризнанные платформы данных. Если такие интеграции не существуют или если эксперт предложит, можно кратко обсудить возможный скрипт для чтения с одной платформы и записи на другую. Такой скрипт должен использовать средства параллельной обработки каждой из платформ. Также можно обсудить стратегии секционирования. На этом шаге входные данные/журналы могут секционироваться по сервису, а выходные данные — по дате. На этой стадии также можно удалить пробельные символы с обоих концов поисковой строки.

11.7.3. Разбиение поисковых строк на слова и другие простые операции

На следующем этапе поисковые строки разбиваются по пробельным символам функцией `split`. Попутно можно рассмотреть такие типичные проблемы, как пропуск пробелов (например, «HelloWorld») или использование других разделителей — точек, дефисов, запятых и т. д. В этой главе предполагается, что такие проблемы встречаются редко и на них можно не обращать внимания. Возможно, вы захотите провести аналитику по журналам поиска, чтобы узнать, насколько распространены эти проблемы. Мы будем называть строки, полученные в результате разбиения, словами поиска. Описание функции `split` в Hive представлено по адресу https://cwiki.apache.org/conluence/display/Hive/LanguageManual+UDF#LanguageManualUDF-StringFunctions, а описание функции `split` в Spark — по адресу https://spark.apache.org/docs/latest/api/sql/index.html#split. На предыдущем этапе мы прочитали данные из файла HDFS, а затем провели разбиение строк.

На этой стадии также можно выполнить различные простые операции, которые вряд ли будут изменяться на протяжении жизни вашей системы, например фильтрацию строк, которые содержат не менее 6 символов и состоят только из букв (то есть без цифр или специальных символов), и преобразование всех строк к нижнему регистру, чтобы не приходилось учитывать регистр символов при дальнейшей обработке. Полученные строки можно записать в другой файл HDFS.

11.7.4. Фильтрация недопустимых слов

Мы рассмотрим следующие составляющие фильтрации допустимых слов (или исключения недопустимых):

1. Управление списками допустимых и недопустимых слов.
2. Проверка списка слов поиска по спискам допустимых и недопустимых слов.

Сервис слов

Сервис слов содержит конечные точки API, возвращающие отсортированные списки допустимых и недопустимых слов. Эти списки занимают не более нескольких мегабайт и сортируются для того, чтобы сделать возможным бинарный поиск. Их малый размер означает, что любой хост, который загружает списки, может кэшировать их в памяти на случай, если сервис слов окажется недоступным. При этом можно использовать типичную горизонтально масштабированную архитектуру для сервиса слов, включающую сервисы бэкенда и пользовательского интерфейса без сохранения состояния, и реплицированный сервис SQL (см. раздел 3.3.2). На рис. 11.5 изображена высокоуровневая диаграмма сервиса слов — простого приложения, которое читает и записывает слова в базу данных SQL. Таблицы SQL для допустимых и недопустимых слов могут содержать строковый столбец для слов и другие столбцы с данными: меткой времени добавления слова в таблицу, указанием пользователя, добавившего слово, и необязательный строковый столбец для примечаний, например, почему слово отнесено к допустимым или недопустимым. Сервис слов предоставляет пользовательский интерфейс для администраторов, в котором можно просмотреть списки допустимых и недопустимых слов и вручную добавить или удалить в них слова, — все эти операции выполняются через конечные точки API. Бэкенд также может предоставлять конечные точки для фильтрации слов по категориям или поиска слов.

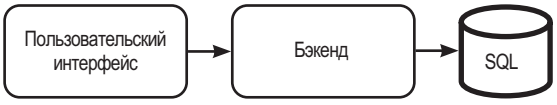

Рис. 11.5. Высокоуровневая архитектура сервиса слов

Исключение недопустимых слов

Пайплайн подсчета слов ETL обращается с запросом к сервису слов за недопустимыми словами, а затем записывает их список в файл HDFS. Возможно, файл HDFS уже сохранен от предыдущего запроса. С тех пор администраторы сервиса слов могли удалить некоторые слова, так что в новом списке могут отсутствовать слова, встречающиеся в старом файле HDFS. HDFS поддерживает только присоединение, так что вы не сможете удалить отдельные слова из файла HDFS; вместо этого придется удалить старый и записать новый файл.

Имея файл HDFS недопустимых слов, можно воспользоваться командой `LOAD DATA` для регистрации таблицы Hive для этого файла, исключить недопустимые слова простым запросом, а затем записать вывод в другой файл HDFS.

Чтобы определить, какие строки поиска содержат недопустимые слова, можно воспользоваться распределенным аналитическим ядром, таким как Spark.

Напишите код на PySpark или Scala или воспользуйтесь запросом Spark SQL для соединения (JOIN) слов пользователя с допустимыми словами.

На собеседовании написание запроса SQL — вернее, эскиза его важнейшей логики — должно занять не более 30 секунд. Вы можете кратко пояснить, что хотите эффективно использовать свои 50 минут и поэтому не хотите тратить драгоценное время на написание идеального запроса SQL. Скорее всего, эксперт согласится, что этот вопрос выходит за рамки собеседования по проектированию систем и вы пришли не для того, чтобы демонстрировать свои навыки SQL, и разрешит двигаться дальше. Исключением может быть собеседование на вакансию инженера данных. В этом случае можно рассмотреть:

- фильтры (например, условия `WHERE`);
- условия `JOIN`;
- агрегатные функции: `AVG`, `COUNT`, `DISTINCT`, `MAX`, `MIN`, `PERCENTILE`, `RANK`, `ROW_NUMBER` и т. д.

```
SELECT word FROM words WHERE word NOT IN (SELECT word from inappropriate_
words);
```

Так как таблица недопустимых слов мала, можно воспользоваться *сопоставляющим соединением* (map join) (соединением, выполняемым сопоставителями в задании MapReduce). За информацией обращайтесь к https://cwiki.apache.org/conluence/display/hive/languagemanual+joins:

```
SELECT /*+ MAPJOIN(i) */ w.word FROM words w LEFT OUTER JOIN inappropriate_
words i ON i.word = w.word WHERE i.word IS NULL;
```

Широковещательное хеш-соединение (broadcast hash join) в Spark аналогично сопоставляющему соединению в Hive. Широковещательное хеш-соединение выполняется между небольшой переменной или таблицей, которая может поместиться в памяти каждого узла (в Spark задается свойством `spark.sql.autoBroadcastJoinThreshold`, по умолчанию 10 Мбайт), и большей таблицей, которая должна быть разделена между узлами. Широковещательное хеш-соединение выполняется следующим образом:

1. Создание хеш-таблицы с меньшей таблицей, в которой ключом является значение, участвующее в соединении, а значением — вся строка данных. Например, в нашем случае соединение устанавливается по строке слова, так что хеш-таблица недопустимых слов со столбцами `word`, `created_at`, `created_by` и т. д. может содержать такие элементы, как {(«apple", ("apple", 1660245908, "brad")), ("banana", ("banana", 1550245908, "grace")), ("orange", ("orange", 1620245107, "angelina")) . . . }.
2. Широковещательная рассылка/копирование этой хеш-таблицы по всем узлам, выполняющим операцию соединения.

3. Каждый узел соединяет (JOIN) меньшую таблицу с частью большей таблицы, находящейся на узле.

Если обе таблицы не помещаются в памяти, выполняется соединение с перестановочной сортировкой слиянием: оба набора данных перемешиваются, записи сортируются по ключу, и выполняется сортировка слиянием, при которой обе стороны последовательно перебираются и соединяются по ключу соединения. При таком подходе не нужно вести статистику по недопустимым словам. Некоторые ресурсы с дополнительной информацией о соединениях в Spark:

- https://spark.apache.org/docs/3.3.0/sql-performance-tuning.html#join-strategy-hints-for-sql-queries или https://spark.apache.org/docs/3.3.0/rdd-programming-guide.html#broadcast-variables. Официальная документация Spark, касающаяся стратегий JOIN в Spark, направленных на улучшение производительностя JOIN. В ней указываются разные стратегии JOIN, но механизмы их реализации подробно не рассматриваются. За этой информацией обращайтесь к ресурсам, перечисленным ниже.
  - https://spark.apache.org/docs/3.3.0/sql-performance-tuning.html#join-strategy-hints-for-sql-queries
  - Дж. Дамий и др. (Damiji, J. et al). A Family of Spark Joins. В книге «Learning Spark, 2nd Edition». O'Reilly Media, 2020.
  - Б. Чамберс, М. Джойнс Захария (Chambers, B., Zaharia, M. Joins). «Spark: The Deinitive Guide: Big Data Processing Made Simple». O'Reilly Media, 2018.
  - https://docs.qubole.com/en/latest/user-guide/engines/hive/hive-mapjoin-options.html
  - https://towardsdatascience.com/strategies-of-spark-join-c0e7b4572bcf

11.7.5. Нечеткое сопоставление и проверка орфографии

Последним шагом обработки перед подсчетом слов должно стать исправление опечаток в словах поиска. Можно написать функцию, которая получает строку, использует алгоритм нечеткого сопоставления для исправления возможных опечаток и возвращает либо исходную строку, либо строку, полученную в результате нечеткого сопоставления. (Нечеткое сопоставление, также называемое приближенным сопоставлением строк, — метод поиска строк, достаточно близких к образцу. Описание алгоритма нечеткого сопоставления выходит за рамки книги.) Затем можно использовать Spark для параллельного выполнения этой функции по списку слов, разделенному на части одинакового размера, с последующей записью вывода в HDFS.

Шаг проверки орфографии выделен в независимую задачу/стадию, потому что существует несколько разных алгоритмов нечеткого сопоставления, библиотек или сервисов, из которых вы можете выбрать нужный алгоритм для оптимизации под ваши требования. Выделение этого этапа позволяет легко переключаться

между библиотекой или сервисом для нечеткого сопоставления, так как изменения в этой стадии пайплайна не повлияют на другие стадии. Если вы используете библиотеку, возможно, ее придется обновлять в соответствии с изменяющимися тенденциями и появлением новых популярных слов.

11.7.6. Подсчет слов

Теперь все готово к подсчету слов. Это может быть простая операция MapReduce, либо можно воспользоваться таким алгоритмом, как Count-Min Sketch (см. раздел 17.7.1).

Приведенный ниже код Scala реализует метод MapReduce. Он слегка изменен по сравнению с версией https://spark.apache.org/examples.html. Слова из входящего файла HDFS сопоставляются с парами `(String, Int)`, которые называются `counts`, сортируются по убыванию счетчиков, а затем сохраняются в другом файле HDFS:

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.map(word => (word, 1)).reduceByKey(_ + _).map(item =>
item.swap).sortByKey(false).map(item => item.swap)
counts.saveAsTextFile("hdfs://...")
```

11.7.7. Фильтр допустимых слов

Шаг подсчета слов должен значительно сократить количество фильтруемых слов. Фильтрация допустимых слов очень напоминает исключение недопустимых слов, раздел 11.7.4.

Можно воспользоваться простой командой Hive, например `SELECT word FROM counted_words WHERE word IN (SELECT word FROM appropriate_words)`, для фильтрации допустимых слов, или решением с сопоставляющим соединением, или широковещательным хеш-соединением вида `SELECT /*+ MAPJOIN(a) */ c.word FROM counted_words c JOIN appropriate_words a on c.word = a.word;`.

11.7.8. Управление новыми популярными неизвестными словами

После подсчета слов на предыдущем шаге в первой сотне могут оказаться новые популярные слова, ранее неизвестные. На этой стадии такие слова записываются в сервис слов, который сохраняет их в таблице SQL `unknown_words`. По аналогии с разделом 11.7.4 сервис слов предоставляет пользовательский интерфейс и конечные точки бэкенда, чтобы члены группы эксплуатации могли вручную добавить эти слова в списки допустимых или недопустимых слов.

Как показано на DAG пакетного задания ETL для подсчета слов на рис. 11.4, этот шаг можно выполнять независимо и параллельно с фильтрацией допустимых слов.

11.7.9. Генерация и передача взвешенного префиксного дерева

Теперь у нас имеется список подходящих слов для построения взвешенного префиксного дерева. Этот список занимает всего несколько мегабайт, так что взвешенное префиксное дерево можно сгенерировать на одном хосте. Алгоритм построения взвешенного дерева выходит за рамки собеседования по проектированию систем. Он скорее будет предметом обсуждения на собеседовании по программированию. Ниже приведено частичное определение класса Scala, но код необходимо писать на языке вашего бэкенда:

```
class TrieNode(var children: Array[TrieNode], var weight: Int) {
  // Функции для:
  // - создания и возвращения узла Trie.
  // - вставки узла в Trie.
  // - получения дочернего узла с наибольшим весом.
}
```

Взвешенное префиксное дерево сериализуется в JSON. Дерево занимает несколько мегабайт. Это может быть слишком много, чтобы разметка загружалась на сторону клиента каждый раз, когда для пользователя выводится панель поиска, но достаточно мало для репликации на всех хостах. Префиксное дерево можно записать в общее хранилище объектов (такое, как AWS S3) или документную базу данных (такую, как MongoDB или Amazon DocumentDB). Хосты бэкенда можно настроить на отправку ежедневных запросов к хранилищу объектов и выборки обновленной строки JSON. Хосты могут отправлять запросы в случайное время, или же их можно настроить на отправку запросов в одно время с небольшим случайным смещением, чтобы избежать перегрузки хранилища объектов большим количеством одновременных запросов.

Если общий объект слишком велик (например, занимает несколько гигабайт), его стоит поместить в CDN. Другое преимущество компактного префиксного дерева заключается в том, что пользователь может загрузить все дерево при загрузке поискового приложения, так что поиск по префиксному дереву будет выполняться на стороне клиента, а не на стороне сервера. Такое решение заметно сокращает количество запросов к бэкенду, что имеет целый ряд преимуществ:

- Если сеть работает ненадежно или медленно, пользователь может не получить некоторые рекомендации при вводе строки поиска, что снижает качество взаимодействия.
- При обновлении префиксного дерева пользователь, который в этот момент вводит строку поиска, может заметить изменения. Например, если строки в старом префиксном дереве были связаны, в новом префиксном дереве эти отношения могут отсутствовать, и пользователь заметит, что предлагаемые варианты отличаются от предыдущих.

Если база пользователей географически распределена, из-за требований высокой производительности сетевая задержка становится недопустимой. Решение развернуть хосты в нескольких датацентрах может оказаться дорогостоящим и создать задержку репликации. CDN — выбор, эффективный с точки зрения затрат.

Сервис автозаполнения должен предоставить конечную точку PUT для обновления его взвешенного префиксного дерева, которое на этом этапе будет использоваться для передачи сгенерированного взвешенного префиксного дерева сервису автозаполнения.

11.8. ВЫБОРКА

Если автозаполнение не требует высокой точности, можно воспользоваться методом выборки, чтобы большинство операций генерации взвешенного префиксного дерева могло выполняться на одном хосте. У такого решения много преимуществ:

- Префиксное дерево генерируется намного быстрее.
- Так как префиксное дерево генерируется быстрее, изменения в коде проще тестировать перед их развертыванием в рабочей среде. Это упрощает разработку, отладку и обслуживание системы в целом.
- Существенно снижается потребление аппаратных ресурсов, включая вычислительные мощности, пространство хранения данных и сеть.

Выборка может применяться на большинстве этапов:

1. Выборка поисковых строк из сервиса ведения журналов. Такой подход обеспечивает наименьшую точность, но и сложность у него наименьшая. Выборка для получения статистически значимого количества слов, содержащих не менее 6 символов, должна быть большой.
2. Выборка слов после разбиения поисковых строк на отдельные слова и фильтрация слов, содержащих не менее 6 символов. Такой подход позволяет избежать вычислительных затрат на фильтрацию допустимых слов, и возможно, вам не понадобится такая большая выборка, как в предыдущем варианте.
3. Выборка слов после фильтрации допустимых слов. Это решение обеспечивает наивысшую точность, но обладает наибольшей сложностью.

11.9. ТРЕБОВАНИЯ К ОБЪЕМУ ХРАНИЛИЩА ДАННЫХ

На основании высокоуровневой архитектуры можно создавать таблицы со столбцами, перечисленными ниже, используя каждую таблицу для заполнения следующей:

1. Необработанный поисковый запрос с меткой времени, идентификатором пользователя и строкой поиска. Таблица может использоваться для многих

других целей, помимо автозаполнения (например, аналитики для выявления интересов пользователя и отслеживания трендов критериев поиска).

2. После разбиения низкоуровневых поисковых строк к таблице со столбцами для даты и слова могут добавляться отдельные слова.
3. Определение того, какие поисковые строки являются словарными словами, и генерация таблицы с датой (копируется из предыдущей таблицы), идентификатором пользователя и словарным словом.
4. Агрегирование словарных слов в таблицу счетчиков слов.
5. Создание взвешенного префиксного дерева для предоставления рекомендаций автозаполнения.

Оценим объем необходимого пространства хранения данных. Допустим, у вас миллиард пользователей; каждый пользователь ежедневно отправляет 10 поисковых запросов, каждый из которых содержит в среднем 20 символов. Каждый день накапливается приблизительно 1 000 000 000 × 10 × 20 = 200 Гбайт поисковых строк. Старые данные могут удаляться один раз в месяц, так что в любой момент времени у вас хранятся данные за 12 месяцев, а журнал поиска будет занимать 200 Гбайт × 365 = 73 Тбайт только для столбца поисковой строки. Чтобы сократить затраты на хранение, можно рассмотреть несколько вариантов решений.

Первый вариант — пожертвовать точностью, применяя приближенные методы и выборку. Например, можно выбрать и сохранить только 10 % поисковых запросов и сгенерировать префиксное дерево только на этих данных.

Другой способ показан на рис. 11.6, где пакетное задание ETL агрегирует и сводит данные за разные периоды для сокращения объема хранимых данных. На каждой стадии можно перезаписать входные данные сводными. В любой момент времени будут храниться только необработанные данные за один день, данные за 4 недели, обобщенные по неделям, и данные за 11 месяцев, сведенные по месяцам. Требования к объему хранилища данных можно еще сократить, сохраняя только верхние 10–20 % наиболее частых строк из каждого задания сведения.

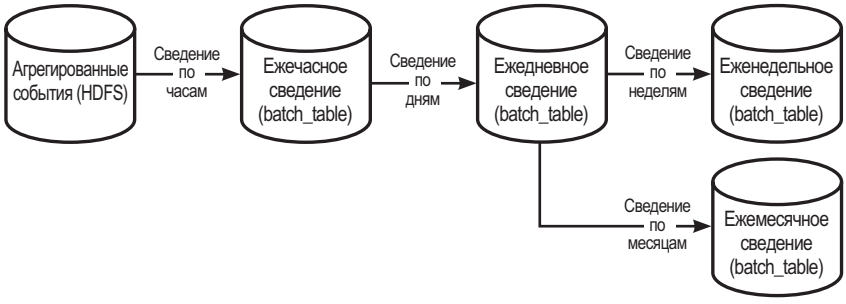

Рис. 11.6. Блок-схема пакетного пайплайна. Задание проводит сведение с возрастающими временными интервалами, чтобы сократить количество строк, обрабатываемых на каждом этапе

Такой подход также улучшает масштабируемость. Без сводного задания пакетному заданию ETL для подсчета слов пришлось бы обрабатывать 73 Тбайт данных, что займет много часов и потребует значительных финансовых затрат. Сводное задание сокращает объем данных, обрабатываемых для завершающего счетчика слов, используемого генератором взвешенных префиксных деревьев.

Для сервиса ведения журналов можно установить короткий период удержания данных (например, 14–30 дней), так что требования к пространству хранения данных составят всего 2,8–6 Тбайт. Ежедневное пакетное задание ETL для генерации взвешенных префиксных деревьев может работать на основе недельных или месячных сводных данных. На рис. 11.7 изображена новая высокоуровневая архитектура со сводными заданиями.

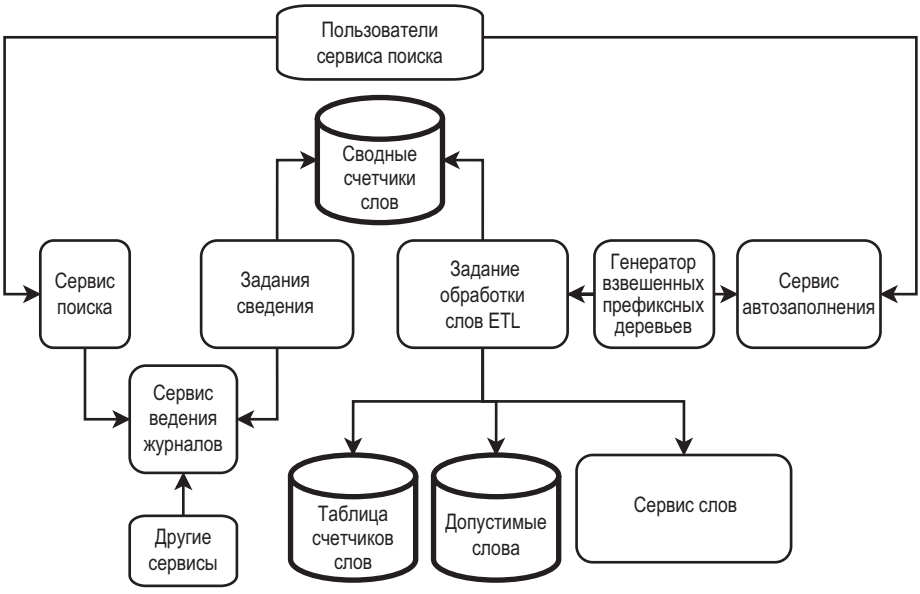

Рис. 11.7. Высокоуровневая архитектура системы автозаполнения со сводными заданиями. Выполняя агрегирование/сведение счетчиков слов по последовательно увеличивающимся интервалам, мы можем сократить требования к объему хранилища и размер кластера задания обработки слов ETL

11.10. ОБРАБОТКА ВЫРАЖЕНИЙ ВМЕСТО ОТДЕЛЬНЫХ СЛОВ

В этом разделе мы обсудим несколько параметров для расширения системы, чтобы она обрабатывала выражения вместо отдельных слов. Префиксное дерево увеличивается, но его все равно можно ограничить несколькими мегабайтами, сохраняя только самые частые выражения.

11.10.1. Максимальная длина рекомендаций автозаполнения

Можно придерживаться ранее принятого решения о том, что рекомендации автозаполнения должны иметь минимальную длину 5 символов. Но какой должна быть максимальная длина? Ее увеличение будет полезно пользователям, но за это придется расплачиваться компромиссами в части затрат и производительности. Системе понадобится больше аппаратных ресурсов или больше времени для записи в журнал и обработки более длинных строк. Префиксное дерево также может стать слишком громоздким.

Необходимо определиться с максимальной длиной. Она будет зависеть от языка и культуры. В нашей системе предполагается только английский язык, но вы должны быть готовы расширить ее для других языков, если это станет функциональным требованием.

Одно из возможных решений — реализация пакетного пайплайна ETL для нахождения 90-го процентиля длины поисковых строк пользователей и использование его значения в качестве максимальной длины. Чтобы вычислить медиану или процентиль, мы сортируем список, а затем выбираем значение в соответствующей позиции. В этой книге мы не будем рассматривать вычисление медианы или процентиля в распределенных системах. Вместо него можно сделать выборку поисковых строк и вычислить 90-й процентиль в ней.

Можно также принять, что аналитика для такого решения будет переусложнением, и применить вместо нее простую эвристику. Начните с 30 символов и изменяйте это значение с учетом обратной связи пользователей, производительности и затрат.

11.10.2. Как исключить недопустимые рекомендации

Нам все еще нужно исключить неприемлемые слова. Для этого можно воспользоваться следующей процедурой:

- Если выражение содержит хотя бы одно недопустимое слово, отфильтровывается все выражение.
- Фильтрация подходящих слов не выполняется, но даются рекомендации автозаполнения для любого слова или выражения.
- Опечатки в словах в выражениях не исправляются. Предполагается, что опечатки встречаются достаточно редко и не появятся в рекомендациях автозаполнения. Также предполагается, что популярные выражения будут в основном написаны правильно, поэтому они будут появляться в рекомендациях автозаполнения.

Трудности связаны с тем, что отфильтровать нужно недопустимые выражения, а не только недопустимые слова. Это комплексная задача, полного решения которой не нашел даже Google (https://algorithmwatch.org/en/auto-completion-disinformation/) из-за необъятности охвата. К числу потенциально недопустимых рекомендаций автозаполнения относятся:

- Дискриминационные или негативные стереотипы, относящиеся к религиозным, гендерным и другим группам.
- Дезинформация, включающая политическую дезинформацию, например теории заговора, относящиеся к изменениям климата или вакцинации, или дезинформация коммерческого плана.
- Клевета на известных личностей или защитников в судебных процессах, в которых вердикт еще не вынесен.

В настоящее время применяются решения, использующие комбинацию эвристики и машинного обучения.

11.11. Ведение журналов, мониторинг и оповещения

Кроме обычных действий из главы 9, следует регистрировать поисковые запросы, которые не возвращают никакие результаты автозаполнения, что указывает на ошибки в генераторе префиксных деревьев.

11.12. Прочие соображения и возможные темы обсуждения

Некоторые требования и темы, о которых может пойти речь в ходе собеседования:

- Существует много часто употребляемых слов длиной более 3 букв — «then», «continue», «hold», «make», «know», «take»[1] и т. д. Некоторые из этих слов стабильно входят в список самых распространенных слов. Подсчет таких слов может стать напрасной тратой вычислительных ресурсов. Может ли система автозаполнения вести список таких слов и использовать приближенные методы для принятия решений о том, какие из них возвращать при вводе текста пользователем?
- Как уже говорилось, журналы пользовательского ввода могут использоваться для других целей помимо автозаполнения, например, для сервиса, который занимается отслеживанием трендов в поисковых запросах для рекомендательных систем.

---

[1] Тогда, продолжить, держать, делать, знать, брать. — *Примеч. пер.*

- Проектирование сервиса распределенного ведения журналов.
- Фильтрация недопустимых поисковых запросов — общий вопрос для большинства сервисов.
- Можно рассмотреть другие входные данные и обработку для персонализации автозаполнения.
- Возможно применение лямбда-архитектуры. Лямбда-архитектура содержит быстрый пайплайн, так что запросы пользователей могут быстро распространяться в генератор взвешенных префиксных деревьев (например, за секунды или минуты); таким образом обеспечивается быстрота обновления рекомендаций автозаполнения за счет точности. Лямбда-архитектура также содержит медленный пайплайн для точных, но более медленных обновлений.
- Корректное сокращение функциональности с возвращением устаревших рекомендаций в случае сбоев на вышележащих компонентах.
- Ограничитель частоты перед сервисом для предотвращения DoS-атак.
- Сервис проверки орфографии имеет отношение к автозаполнению, но отличается от него. Этот сервис предоставляет возможные рекомендации пользователю, который ввел слово с ошибкой. Сервис проверки орфографии может использовать такие экспериментальные методы, как A/B-тестирование или метод многорукого бандита, для оценки влияния разных функций нечеткого сопоставления на отток пользователей.

ИТОГИ

- Автозаполнение — пример системы, которая непрерывно поглощает большой объем данных и перерабатывает их в компактную структуру данных. К такой системе пользователи обращаются с запросами для конкретной цели.
- Спектр практического применения автозаполнения очень широк. Автозаполнение может быть общим сервисом, используемым многими другими сервисами.
- Автозаполнение отчасти накладывается на функциональность поиска, но эти инструменты используются для разных целей. Поиск предназначен для нахождения документов, тогда как автозаполнение должно предлагать текст, который собирается ввести пользователь.
- Система автозаполнения выполняет большой объем предварительной обработки данных, так что необходимо выделить предварительную обработку и запросы в разные компоненты, разрабатываемые и масштабируемые независимо друг от друга.

- В качестве источников данных для сервиса автозаполнения могут использоваться сервис поиска и сервис ведения журналов. Сервис автозаполнения может обрабатывать поисковые строки, которые эти сервисы получают от пользователей, и предлагать рекомендации автозаполнения на основании этих строк.
- Для автозаполнения используется взвешенное префиксное дерево. В этом случае поиск выполняется быстро, а затраты памяти невелики.
- Большое задание агрегирования разбивается на несколько этапов для сокращения затрат памяти и вычислительных ресурсов. Компромисс — высокая сложность и трудность обслуживания.
- Прочие соображения включают другие возможности применения обработанных данных, выборку, фильтрацию контента, персонализацию, лямбда-архитектуру, корректное сокращение функциональности и ограничение частоты.

12 Проектирование Flickr

В ЭТОЙ ГЛАВЕ

- ✓ Выбор сервисов хранения данных в зависимости от нефункциональных требований
- ✓ Ограничение доступа к критическим сервисам
- ✓ Использование саг для асинхронной обработки

В этой главе мы спроектируем сервис фотохостинга наподобие Flickr. Кроме размещения файлов/изображений, пользователи сервиса могут добавлять метаданные к файлам и другим пользователям, например токены управления доступом, комментарии или пометки избранного.

Публикация и взаимодействие с изображениями и видео — основные виды функциональности практически в каждом приложении социальной сети. Эти вопросы очень часто встречаются на собеседованиях. В этой главе будет рассмотрен дизайн распределенной системы для хостинга и взаимодействия с графическими изображениями на миллиард пользователей — как физических, так и программных. Вы увидите, что задача вовсе не сводится к простому подключению CDN. Вы узнаете, как спроектировать систему для масштабируемых операций предварительной обработки, которые необходимо выполнить с выгруженным контентом, прежде чем он будет готов к загрузке.

12.1. ПОЛЬЗОВАТЕЛЬСКИЕ ИСТОРИИ И ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

Обсудите с экспертом основные пользовательские истории и кратко запишите их:

- Пользователь может просматривать фотографии, опубликованные другими. Будем называть такого пользователя *зрителем*.
- Приложение должно генерировать и отображать миниатюры (thumbnails) шириной 50 пикселей. Пользователь просматривает несколько фотографий, размещенных в сетке, и последовательно выбирает их для просмотра в полном разрешении.
- Пользователь может загружать (upload) фотографии. Будем называть такого пользователя *автором*.
- Автор может назначать уровни управления доступом для своих фотографий. Один из вопросов, которые вы можете задать эксперту, — должно ли управление доступом действовать на уровне отдельных фотографий или же автор может разрешить зрителю просматривать все фотографии либо полностью запретить просмотр. Мы выберем для простоты второй вариант.
- Фотографиям назначаются заранее определенные поля метаданных, значения которых предоставляются автором. Например, в таких полях может храниться информация о месте съемки или теги.
- Пример динамических метаданных — список пользователей, имеющих доступ для чтения к файлу. Метаданные являются динамическими, потому что их можно изменять.
- Пользователи могут оставлять комментарии к фотографиям. Автор может разрешить или запретить комментарии. Пользователь может получать оповещения о новых комментариях.
- Пользователь может добавить фотографию в избранное.
- Пользователь может проводить поиск по названиям и описаниям фотографий.
- Фотографии можно сохранять программным путем. Мы будем считать выражения «просмотреть» и «сохранить» (download) синонимами и не станем обсуждать детали относительно того, могут ли пользователи сохранять фотографии в память устройства.
- Краткое обсуждение персонализации.

Мы не будем затрагивать некоторые моменты:

- Возможность фильтровать фотографии по метаданным. Это требование удовлетворяется простым параметром запроса SQL, поэтому его обсуждать не нужно.

- Метаданные фотографий, записанные клиентом: место съемки (от оборудования — например, GPS), время (от часов устройства) и параметры камеры (от операционной системы).
- Видео. Обсуждение многих параметров видео (например, кодеков) требует специализированных знаний, выходящих за рамки общего собеседования по проектированию систем.

12.2. НЕФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

Несколько вопросов о нефункциональных требованиях, которые можно обсудить:

- Сколько пользователей и загрузок через API предполагается?
  - Система должна быть масштабируемой. Она должна обслуживать 1 миллиард пользователей, распределенных по всему миру. Ожидается интенсивный трафик. Предполагается, что 1 % пользователей (10 миллионов) ежедневно выгружает 10 изображений в высоком разрешении (10 Мбайт). Это преобразуется в 1 петабайт выгрузок ежедневно, или 3,65 экзабайт за 10 лет. Средний трафик составляет около 1 Гбайт/с, но возможны выбросы трафика, так что стоит планировать 10 Гбайт/с.
- Должны ли фотографии становиться доступными сразу после загрузки? Должно ли удаление происходить немедленно? Должны ли изменения настроек конфиденциальности вступать в силу немедленно?
  - Фотографии должны быть доступны для всей пользовательской базы в течение нескольких минут. Применительно к фотографиям — и комментариям — можно пожертвовать некоторыми нефункциональными характеристиками (такими, как консистенстность или задержка) для снижения затрат. Консистентность в конечном счете приемлема.
  - Настройки конфиденциальности должны вступать в действие быстрее. Не обязательно обеспечивать физическое удаление определенной фотографии из всех хранилищ за несколько минут; допустима задержка в несколько часов. Тем не менее она должна стать недоступной для всех пользователей в течение нескольких минут.
- Фотографии в высоком разрешении требуют высокой скорости передачи данных по сети, что может повлечь высокие затраты. Как их ограничить?
  - После краткого обсуждения вы приходите к соглашению, что пользователь должен иметь возможность выгружать (то есть сохранять) только одно фото в высоком разрешении за раз, но несколько миниатюр в низком разрешении можно выгружать одновременно. При выгрузке файлы должны передаваться по одному.

Другие нефункциональные требования:

- Высокая доступность — например, «пять девяток». Не должно быть простоев, во время которых пользователи не могут загружать или сохранять фотографии.
- Высокая производительность и низкая задержка при значении 99-го процентиля 1 секунда для выгрузки миниатюр; не требуется для фотографий высокого разрешения.
- Высокая производительность для выгрузки не требуется.

Небольшое замечание по поводу миниатюр (thumbnails): для отображения миниатюр изображений в полном разрешении можно воспользоваться тегом CSS `img` с атрибутами `width` (https://developer.mozilla.org/en-US/docs/Web/HTML/Element/img#attr-width) или `height`. В мобильных приложениях имеются похожие теги разметки. Такой подход повышает нагрузку на сеть и плохо масштабируется. Чтобы вывести сетку миниатюр на стороне клиента, необходимо скачивать каждое изображение в полном разрешении. Можно предложить эксперту реализовать эту возможность в MVP (минимальной жизнеспособной версии продукта). Когда вы займетесь масштабированием сервиса для обслуживания интенсивного трафика, можно будет рассмотреть два способа генерации миниатюр.

Первый — генерация миниатюры на сервере по изображению в высоком разрешении каждый раз, когда клиент запросит миниатюру. Такое решение будет масштабируемым, если генерация миниатюры обходится дешево по вычислительным ресурсам. Тем не менее файл изображения в полном разрешении занимает десятки мегабайт. Зритель обычно запрашивает сетку с > 10 миниатюрами за раз. Если предположить, что изображение в полном разрешении занимает 10 Мбайт (хотя оно может быть и намного больше), то для выполнения требования односекундной задержки 99-го процентиля сервису придется обрабатывать > 100 Мбайт данных менее чем за 1 секунду. Более того, за несколько секунд прокрутки серии миниатюр зритель может отправить много таких запросов. Такое решение может быть приемлемым по вычислениям, если хранение и обработка выполняются на одной машине и на ней стоят диски SSD вместо традиционных жестких дисков. Однако затраты будут непомерными. Более того, вы не сможете выполнить функциональное секционирование обработки и хранения на отдельные сервисы. Сетевая задержка при ежесекундной пересылке нескольких гигабайт между сервисами обработки и хранения не позволит обеспечить 1-секундное значение 99-го процентиля. Таким образом, этот способ в целом нам не подходит.

Единственное масштабируемое решение — сгенерировать и сохранить миниатюру сразу же после загрузки файла и предоставлять эти миниатюры, когда зритель будет их запрашивать. Каждая миниатюра занимает несколько килобайт, так что затраты на хранение данных будут небольшими. Также можно кэшировать как миниатюры, так и файлы изображений в высоком разрешении на стороне клиента (см. раздел 12.7). В этой главе мы обсудим именно такое решение.

12.3. ВЫСОКОУРОВНЕВАЯ АРХИТЕКТУРА

На рис. 12.1 изображена исходная высокоуровневая архитектура. Как авторы, так и зрители отправляют запросы на загрузку или сохранение файлов фотографий через бэкенд. Бэкенд взаимодействует с сервисом SQL.

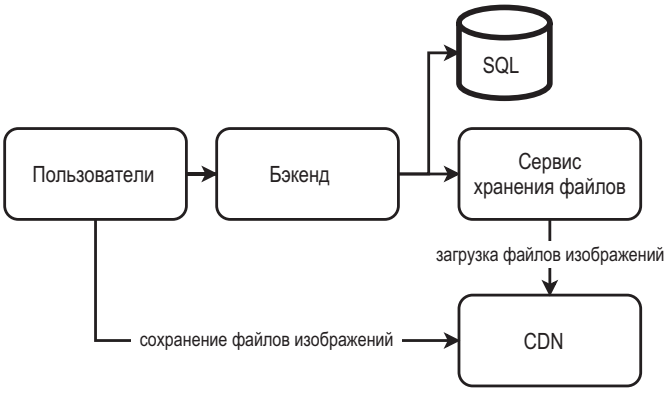

Рис. 12.1. Исходная высокоуровневая архитектура сервиса фотохостинга. Пользователи могут сохранять файлы изображений прямо из CDN. Загрузка в CDN может буферизоваться через отдельный распределенный сервис хранения файлов. Другие данные (информация пользователя, разрешения доступа к файлу изображения) могут храниться в SQL

Первый элемент — CDN для хранения файлов изображений и метаданных изображений (метаданные представляют собой отформатированные строки JSON или YAML). Скорее всего, это будет сторонний сервис.

По причинам, включающим перечисленные ниже, вам также может понадобиться отдельный сервис распределенного хранения файлов, чтобы авторы могли отправлять свои фотографии; этот сервис будет обеспечить взаимодействие с CDN. Будем называть его сервисом хранения файлов.

- В зависимости от условий SLA с провайдером CDN, репликация изображений по разным датацентрам может занять до нескольких часов. В это время сохранение (выгрузка) изображений может занимать чуть больше времени, особенно при высокой частоте запросов на сохранение, потому что многие зрители хотят получить этот файл.
- Задержка загрузки файлов изображений в CDN может оказаться недопустимо большой для автора. Возможно, CDN не сможет поддерживать одновременную загрузку изображений многими авторами. Сервис хранения файлов можно масштабировать по мере необходимости для обработки высокого трафика загрузки/записи.

- После загрузки файлов в CDN можно либо удалить их из сервиса хранения, либо сохранить как резервную копию. Второй вариант можно выбрать, если вы не полностью доверяете соглашению SLA для CDN и хотите использовать сервис хранения файлов как резервную копию на случай, если CDN утратит работоспособность. Кроме того, CDN может иметь период удержания данных в несколько недель или месяцев, после чего файлы удаляются и при необходимости загружаются снова из указанного источника. Среди других возможных ситуаций можно выделить неожиданное требование об отключении CDN из-за проблем с безопасностью.

12.4. Схема SQL

Для динамических данных, которые отображаются в клиентских приложениях, например информации о том, какие фотографии связаны с тем или иным пользователем, используется база данных SQL. Для таких данных можно определить схему таблицы SQL, приведенную в листинге 12.1. Таблица Image содержит метаданные изображения. Каждому автору можно назначить собственный каталог CDN, который отслеживается в таблице ImageDir. Описания схем включаются в команду `CREATE`.

Листинг 12.1. Команда SQL CREATE для таблиц Image и ImageDir

```
CREATE TABLE Image (
cdn_path VARCHAR(255) PRIMARY KEY COMMENT="Путь к файлу изображения
➥ в CDN.",
cdn_photo_key VARCHAR(255) NOT NULL UNIQUE COMMENT="Идентификатор,
➥ назначенный CDN.",
file_key VARCHAR(255) NOT NULL UNIQUE COMMENT="Идентификатор, назначенный
➥ сервису хранения файлов. Этот столбец не понадобится, если вы удаляете
➥ изображение из сервиса хранения файлов после его выгрузки в CDN.",
resolution ENUM('thumbnail', 'hd') COMMENT="Признак миниатюры
➥ или изображения в высоком разрешении",
owner_id VARCHAR(255) NOT NULL COMMENT="Идентификатор пользователя,
➥ который является владельцем изображения.",
is_public BOOLEAN NOT NULL DEFAULT 1 COMMENT="Указывает, является ли
➥ изображение общедоступным или приватным.",
INDEX thumbnail (Resolution, UserId) COMMENT="Составной индекс разрешения
➥ и идентификатора пользователя позволяет быстро найти миниатюру
➥ или изображение в высоком разрешении, принадлежащее конкретному пользователю."
) COMMENT="Метаданные изображения.";

CREATE TABLE ImageDir (
cdn_dir VARCHAR(255) PRIMARY KEY COMMENT="Каталог CDN, назначенный
➥ пользователю.",
user_id INTEGER NOT NULL COMMENT="Идентификатор пользователя."
) COMMENT="Хранит каталог CDN, назначенный каждому пользователю.";
```

Так как выборка фотографий по идентификатору пользователя и разрешению представляет собой типичный запрос, таблицы будут индексироваться по этим

полям. После этого можно использовать методы, рассмотренные в главе 4, для масштабирования чтения из таблиц SQL.

Чтобы дать автору право предоставлять разрешения зрителям на просмотр фотографий и на добавление их в «Избранное», можно определить схему, приведенную в листинге ниже. Другой вариант — использование двух таблиц для определения логического столбца `is_favorite` таблицы Share, но за это придется расплачиваться появлением разреженного столбца, использующего лишнее пространство хранилища:

```
CREATE TABLE Share (
  id            INT PRIMARY KEY
  cdn_photo_key VARCHAR(255),
  user_id       VARCHAR(255)
);

CREATE TABLE Favorite (
  id INT PRIMARY KEY
  cdn_photo_key VARCHAR(255) NOT NULL UNIQUE,
  user_id VARCHAR(255) NOT NULL UNIQUE
);
```

12.5. СТРУКТУРА КАТАЛОГОВ И ФАЙЛОВ В CDN

Обсудим один из возможных вариантов организации каталогов CDN. Иерархия каталогов может иметь вид «пользователь → альбом → разрешение → файл». Также можно включить дату, поскольку пользователя могут больше интересовать недавние файлы.

Каждому пользователю выделяется собственный каталог CDN. Можно разрешить пользователю создавать альбомы, каждый из которых содержит 0 или больше фотографий. Между альбомами и фотографиями существует связь типа «один ко многим»; другими словами, каждая фотография может принадлежать только одному альбому. В CDN можно размещать фотографии не в созданных альбомах, а в альбоме с именем default. Таким образом, каталог пользователя может содержать один или несколько каталогов альбомов.

Каталог альбома может содержать несколько файлов изображений с разным разрешением, каждый в отдельном каталоге, и файл метаданных изображения в формате JSON. Например, в каталоге original может содержаться изначально выгруженный файл swans.png, а в каталоге thumbnail — сгенерированная миниатюра swans_thumbnail.png.

Значение CdnPath строится по шаблону <имя_альбома>/<разрешение>/<имя_изображения.расширение>. Идентификатор или имя пользователя не нужны, потому что эта информация содержится в поле UserId.

Например, пользователь с именем «alice» может создать альбом с именем «nature», в котором размещается изображение с именем «swans.png». Значение CdnPath

будет иметь вид nature/original/swans.png. Соответствующая миниатюра будет иметь значение CdnPath nature/thumbnail/swans_thumbnail.png. Команда `tree` в CDN выведет информацию, приведенную ниже (bob — имя другого пользователя):

```
$ tree ~ | head -n 8
.
├── alice
│   └── nature
│       ├── original
│       │   └── swans.png
│       └── thumbnail
│           └── swans_thumbnail.png
├── bob
```

Далее термины «изображение» и «файл» будут использоваться как синонимы.

12.6. ЗАГРУЗКА ФОТОГРАФИИ

Где должны генерироваться миниатюры — на стороне клиента или на стороне сервера? Как отмечалось во введении, следует быть готовым к тому, что вам придется обсуждать разные варианты и оценивать их компромиссы.

12.6.1. Генерация миниатюр на стороне клиента

Генерация миниатюр на стороне клиента экономит вычислительные ресурсы бэкенда, а миниатюра занимает мало памяти, так что она не оказывает особого влияния на сетевой трафик. Миниатюра размером 100 пикселей занимает около 40 Кбайт — незначительная прибавка к размеру фотографий в высоком разрешении, которые могут занимать от нескольких мегабайт до десятков мегабайт.

До начала загрузки клиент может проверить, не была ли миниатюра уже загружена в CDN. Во время загрузки выполняются следующие действия, показанные на рис. 12.2:

1. Генерация миниатюры.
2. Оба файла помещаются в папку, после чего сжимаются алгоритмом Gzip или Brotli. Сжатие обеспечивает значительную экономию сетевого трафика, но бэкенду придется расходовать ресурсы процессора и памяти на распаковку каталога.
3. Использование запроса POST для выгрузки сжатого файла в каталог CDN. Тело запроса представляет собой строку JSON с описанием количества и разрешения загружаемых изображений.
4. Содание в CDN необходимых каталогов, распаковка сжатого файла и запись файлов на диск. Репликация по другим датацентрам (см. следующий вопрос).

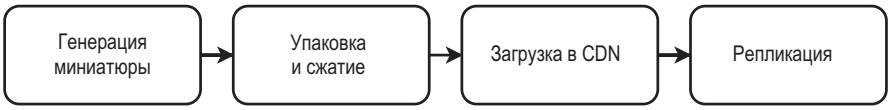

Рис. 12.2. Загрузка фотографии на фотохостинг со стороны клиента

ПРИМЕЧАНИЕ Как отмечалось во введении, на собеседовании никто не будет ожидать от вас исчерпывающих знаний алгоритмов сжатия, криптографически безопасных алгоритмов хеширования, алгоритмов аутентификации, конвертирования пикселей в мегабайты и даже самого термина «миниатюра». От вас требуется умение рассуждать логично и четко излагать свои мысли. Эксперт предполагает, что вы сможете объяснить, что миниатюры меньше изображений в высоком разрешении, что сжатие пригодится для передачи больших файлов по сети, а механизмы аутентификации и авторизации необходимы для файлов и пользователей. Если вы не знаете термина «миниатюра», используйте описания типа «сетки уменьшенных фотографий» или «маленькие фотографии», объясните, что имеется в виду небольшое количество пикселей и, как следствие, небольшой размер файла.

Недостатки генерации на стороне клиента

Обработка на стороне клиента имеет ряд неочевидных недостатков. Вы не контролируете устройство клиента и не знаете особенностей его среды, что усложняет воспроизведение ошибок. В коде также придется предусмотреть гораздо больше сценариев сбоев, которые могут происходить на стороне клиента (по сравнению с сервером). Например, обработка изображения может завершиться ошибкой, потому что у клиента кончилось место на жестком диске или из-за внезапного сбоя сетевого соединения.

Многие из таких ситуаций, возникающих на стороне клиента, вы не контролируете. Вы можете упустить часть сценариев отказов в ходе реализации и тестирования, а это усложнит отладку — будет труднее воспроизвести ситуацию, возникающую на клиентском устройстве, а не на принадлежащем вам сервере, к которому вы имеете доступ уровня администратора.

На работу приложения могут повлиять многие факторы, из-за которых труднее определить, какую же информацию нужно сохранять в журнале:

- Генерация на стороне клиента требует реализации и обслуживания кода генерации миниатюр для каждого типа клиента (браузер, Android, iOS). Все они используют разные языки, если только вы не воспользуетесь кроссплатформенными фреймворками, такими как Flutter и React Native, у которых есть свои недостатки.
- Аппаратные факторы, например недостаточно быстрый процессор или нехватка памяти, могут привести к тому, что генерация миниатюр станет недопустимо медленной.

- Конкретная версия операционной системы на стороне клиента может содержать ошибки или проблемы безопасности, из-за которых обработка изображений может быть слишком рискованной или создать проблемы, которые очень трудно предвидеть иди диагностировать. Например, если в ОС произойдет внезапный сбой во время выгрузки изображений, файлы могут быть повреждены.
- Другие программы, работающие на стороне клиента, могут потреблять слишком много ресурсов процессора или памяти, в результате чего генерация миниатюр завершится неудачей или миниатюры будут генерироваться слишком медленно. На стороне клиента также могут работать вредоносные программы (например, вирусы), способные помешать работе приложения. Наличие таких программ на клиенте проверять нецелесообразно, и нельзя гарантировать, что клиенты следуют передовым практикам безопасности.
- К сказанному выше: мы можем следовать передовым практикам безопасности на своих системах, чтобы защититься от вредоносных программ, но повлиять на клиенты, чтобы они делали то же самое, нельзя. По этой причине необходимо минимизировать объем хранимых данных и обработку на стороне клиента и хранить и обрабатывать данные только на стороне сервера.
- На выгрузку файлов может влиять конфигурация сети на стороне клиента: заблокированные порты, хосты, возможно — VPN и т. д.
- Некоторые из клиентов могут использовать ненадежное сетевое соединение. Возможно, понадобится логика на случай неожиданного разрыва соединения: допустим, клиент сохраняет сгенерированную миниатюру на своем устройстве, прежде чем выгружать ее на сервер. В случае неудачной выгрузки клиенту не придется заново генерировать миниатюры, прежде чем повторять попытку.
- К сказанному выше: на устройстве может не хватать памяти для сохранения миниатюр. Необходимо помнить, что в реализации клиента нужно проверить наличие свободной памяти перед генерацией миниатюры; в противном случае автору придется ждать, пока будет сгенерирована миниатюра, а потом он столкнется с ошибкой из-за нехватки памяти.
- В связи с тем же пунктом: генерация миниатюр на стороне клиента может привести к тому, что приложению понадобится больше разрешений — например, разрешение записи в локальное хранилище. Даже если вы не будете злоупотреблять этим разрешением, система может быть взломана изнутри или извне, и взломщики смогут выполнять вредоносные операции на устройствах пользователей.

Практическая проблема заключается в том, что каждый отдельный недостаток может влиять лишь на небольшую группу пользователей, и вы решите, что

тратить ресурсы на его исправление неэффективно. С другой стороны, в совокупности эти недостатки могут влиять на заметную часть потенциальной базы пользователей.

Более долгий и рутинный цикл выпуска

Так как обработка на стороне клиента повышает риск ошибок и затраты на их исправление, придется пожертвовать значительные ресурсы и время на тестирование каждой итерации перед развертыванием, что замедлит разработку. Вы не сможете воспользоваться непрерывной интеграцией / непрерывным развертыванием (CI/CD), как при разработке сервисов. Вам придется перейти на жизненный цикл выпуска, показанный на рис. 12.3. В нем каждая новая версия вручную тестируется внутренними пользователями, а затем постепенно развертывается на растущей части пользовательской базы. Вы не сможете быстро выпускать и отменять небольшие изменения. Так как процесс выпуска становится медленным и рутинным, каждый релиз будет содержать много изменений кода.

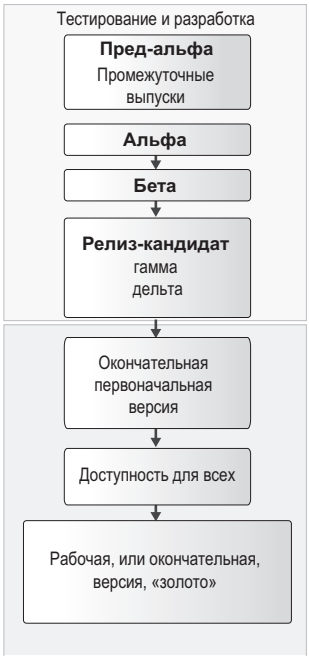

Рис. 12.3. Пример жизненного цикла выпуска программного продукта. Новая версия вручную тестируется внутренними пользователями в альфа-фазе, после чего эта версия передается группе пользователей, увеличивающейся на каждом последующем этапе. Жизненные циклы выпуска в некоторых компаниях содержат больше этапов, чем показано на диаграмме. Источник: Heyinsun (https://commons.wikimedia.org/w/index.php?curid=6818861), CC BY 3.0 (https://creativecommons.org/licenses/by/3.0/deed.en)

В ситуации, когда выпуск / развертывание изменений в коде (на стороне клиента или на стороне сервера) выполняется медленно, возможным решением, чтобы избежать ошибок в новом релизе, станет включение нового кода без удаления старого, как показано на рис. 12.4. В следующем примере предполагается, что мы выпускаем новую функцию, но метод можно обобщить для любого нового кода:

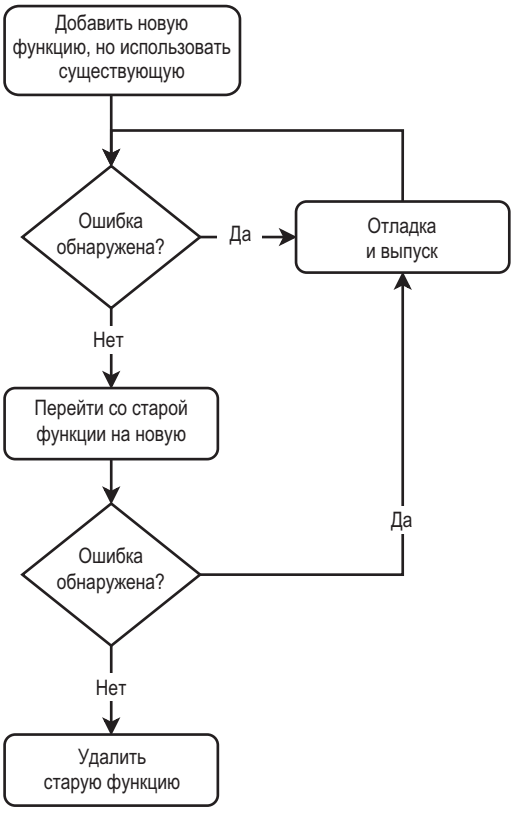

Рис. 12.4. Блок-схема процесса выпуска программного продукта, в котором новый код становится доступным, постепенно замещая старый код в базе

1. Добавьте новую функцию. Запустите новую функцию с такими же исходными данными, как у существующей, но продолжите использовать вывод существующей функции вместо новой. Заключите использование новой функции в команды `try-catch`, так что исключение не приведет к сбою приложения. В команде `catch` зарегистрируйте исключение и отправьте данные в сервис ведения журналов, чтобы диагностировать и исправить ошибку.
2. Отладьте функцию и выпускайте новые версии до тех пор, пока ошибки не пропадут.
3. Переведите код со старой функции на новую. Заключите код в блоки `try-catch`, где команда `catch` регистрирует исключение, и используйте старую функцию в качестве резервного варианта в аварийных ситуациях. Выпустите функцию и понаблюдайте за возможными проблемами. Если они возникнут, вернитесь к старой функции (то есть на шаг назад).
4. Когда вы будете уверены, что новая функция достаточно надежна (полной уверенности в отсутствии ошибок быть не может), удалите старую функцию из кода. Тем самым вы упростите чтение и обслуживание кода.

12.6.2. Генерация миниатюр на бэкенде

Мы обсудили недостатки генерации миниатюр на стороне клиента. Генерация на сервере требует больших аппаратных ресурсов, а также усилий инженеров для создания и обслуживания сервиса бэкенда, но сервис может создаваться на том же языке и с теми же инструментами, что и другие сервисы. Вы можете решить, что приобретения не оправдывают затрат.

В этом разделе обсуждается процесс генерации миниатюр на бэкенде. Он состоит из трех основных этапов:

1. Перед загрузкой файла следует проверить, не загружался ли он ранее. Это позволит избежать дорогостоящих загрузок дубликатов.
2. Файл загружается в сервис хранения данных и CDN.
3. Миниатюра генерируется и загружается в сервис хранения файлов и CDN.

Когда файл загружается на бэкенд, последний записывает его в сервис хранения файлов и CDN, а затем запускает потоковое задание для генерации миниатюры. Основное назначение сервиса хранения файлов — буфер для отправки в CDN, так что можно реализовать репликацию между хостами в датацентре, но не в других датацентрах. В случае серьезного сбоя в датацентре и потери данных вы также сможете использовать файлы из CDN для операций восстановления. Сервис хранения файлов и CDN используются как резервные копии друг для друга.

Для масштабирования загрузки файлов изображений некоторые этапы могут быть асинхронными, что позволяет применить решение на базе саги. О том, что такое сага, см. в разделе 5.6.

Решение с хореографической сагой

На рис. 12.5 изображены сервисы и топики Kafka в хореографической саге. Подробное описание приводится ниже. Номера этапов помечены на обеих диаграммах:

1. Пользователь сначала хеширует изображение, а затем отправляет запрос GET к бэкенду, чтобы проверить, не загружалось ли уже изображение. К примеру, пользователь успешно загрузил изображение в предыдущем запросе, но в сетевом соединении произошел сбой, тогда как сервис хранения файлов или бэкенд вернул признак успеха, так что пользователь может повторить попытку загрузки.
2. Бэкенд отправляет запрос к сервису хранения файлов.
3. Сервис хранения файлов возвращает ответ, который сообщает, был ли файл успешно выгружен ранее.
4. Бэкенд возвращает этот ответ пользователю.

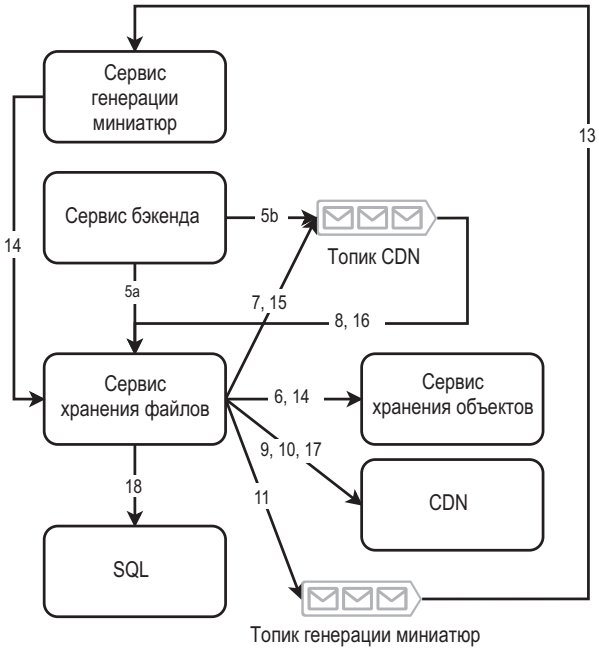

Рис. 12.5. Хореография генерации миниатюр, начиная с этапа 5a. На стрелках указаны номера этапов, описанные в тексте. Для ясности на диаграмме не представлен пользователь. Некоторые события, производимые и потребляемые сервисом хранения файлов в топиках Kafka, указывают ему передавать файлы изображений между сервисом хранения объектов и CDN. Также присутствуют события для запуска генерации миниатюр и для записи метаданных CDN в сервис SQL

5. Этот этап зависит от того, был ли файл успешно загружен:
   a) Если файл не был загружен, пользователь отправляет его сервису хранения файлов через бэкенд. (Пользователь может сжать файл перед отправкой.)
   b) Если же файл уже успешно загружен, бэкенд отправляет событие генерации миниатюры в топик Kafka. После этого можно переходить к шагу 8.
6. Сервис хранения файлов записывает файл в сервис хранения объектов.
7. После успешной записи файла сервис хранения файлов отправляет событие в топик Kafka CDN, после чего возвращает пользователю признак успеха через бэкенд.
8. Сервис хранения файлов потребляет событие с шага 6, которое содержит хеш изображения.
9. По аналогии с шагом 1 сервис хранения файлов отправляет запрос к CDN с хешем изображения, чтобы определить, было ли изображение отправлено в CDN ранее. Это могло случиться, если сервис хранения файлов загружал

файл изображения в CDN, но прежде чем соответствующая контрольная точка была записана в топик CDN, произошел сбой.

10. Сервис хранения файлов загружает файл в CDN. Это делается асинхронно и независимо от загрузки в сервис хранения файлов, так что опыт взаимодействия не изменится, если загрузка в CDN происходит медленно.
11. Сервис хранения файлов отправляет событие генерации миниатюр, содержащее идентификатор файла, в топик Kafka генерации миниатюр и получает ответ с признаком успеха от сервиса Kafka.
12. Бэкенд возвращает пользователю ответ о том, что файл изображения успешно загружен. Он возвращает этот ответ только после создания события генерации миниатюры, чтобы убедиться, что это событие создано; это необходимо, чтобы обеспечить генерацию миниатюры. Если отправка события в Kafka завершается неудачей, пользователь получит ответ 504 («Тайм-аут»). Пользователь может перезапустить процесс, начиная с этапа 1. А что, если событие будет отправлено в топик Kafka несколько раз? Kafka гарантирует, что доставка будет выполнена ровно один раз, следовательно, проблемы не будет.
13. Сервис генерации миниатюр потребляет событие из Kafka, чтобы начать генерацию.
14. Сервис генерации миниатюр получает файл из сервиса хранения файлов, генерирует миниатюры и записывает результат в сервис хранения объектов через сервис хранения файлов.

    Почему сервис генерации миниатюр не осуществляет прямую запись в CDN?

    - Сервис генерации миниатюр должен быть автономным сервисом, который получает запрос на генерацию миниатюры, извлекает файл из сервиса хранения файлов, генерирует миниатюру и записывает результат обратно в сервис хранения файлов. Прямая запись в другие приемники (такие, как CDN) создает дополнительную сложность: например, если CDN в текущее время находится под высокой нагрузкой, сервис генерации миниатюр должен периодически проверять, готов ли сервис CDN к получению файла, и одновременно обеспечить, чтобы свободная память не закончилась. Решение, в котором запись в CDN выполняется под контролем сервиса хранения файлов, получается более простым в разработке и обслуживании.
    - Каждый сервис или хост, которому разрешено записывать в CDN, создает дополнительные затраты на управление безопасностью. Не разрешая сервису генерации миниатюр обращаться к CDN, вы уменьшаете поверхность атаки.
15. Сервис генерации миниатюр записывает ThumbnailCdnRequest в топик CDN, чтобы дать указание сервису хранения файлов записать миниатюры в CDN.

16. Сервис хранения файлов потребляет событие из топика CDN и получает миниатюру из сервиса хранения объектов.
17. Сервис хранения файлов записывает миниатюру в CDN. CDN возвращает ключ файла.
18. Сервис хранения файлов вставляет этот ключ в таблицу SQL (если ключ еще не существует), в которой хранятся соответствия между идентификаторами пользователей и ключами. Учтите, что этапы 16–18 являются блокирующими. Если хост сервиса хранения файлов столкнется со сбоем на этапе вставки, его хост-заместитель продолжит работу с этапа 16. Миниатюра занимает лишь несколько килобайт, так что затраты вычислительных и сетевых ресурсов на повторную попытку пренебрежимо малы.
19. В зависимости от того, насколько быстро CDN сможет предоставить эти файлы изображений (в высоком разрешении и миниатюры), файлы можно удалить из сервиса хранения файлов немедленно или же реализовать периодическое пакетное задание ETL для удаления файлов, созданных час назад. Такое задание также может дать указание CDN обеспечить репликацию файлов по разным датацентрам, прежде чем удалять их из сервиса хранения файлов, но подобное решение может оказаться переусложнением. Сервис хранения файлов может хранить хеши файлов, что позволяет ему отвечать на запросы и проверять, был ли файл выгружен ранее. Можно реализовать пакетное задание ETL для удаления хешей, созданных более часа назад.

Типы транзакций

Какие транзакции являются компенсируемыми, поворотными или транзакциями с возможностью повторения? Этапы, предшествующие этапу 11, являются компенсируемыми транзакциями, потому что на них пользователю не отправляется ответ, подтверждающий успешность загрузки. Этап 11 является поворотной транзакцией, потому что вы подтверждаете пользователю успешность загрузки и повторная попытка не нужна. Этапы 12–16 являются транзакциями с возможностью повторения. У нас имеются необходимые данные (файл изображения) для повторения попыток этих транзакций (генерации миниатюр), что гарантирует их успех.

Если вместо миниатюры и исходного разрешения вы решите сгенерировать несколько изображений с разными разрешениями, компромиссы каждого подхода проявятся сильнее.

Что, если для загрузки фотографий используется FTP вместо HTTP POST или RPC? FTP записывает данные на диск, так что любая дальнейшая обработка будет включать задержку и вычислительные ресурсы для чтения с диска в память. Если вы загружаете сжатые файлы, то чтобы восстановить файл, необходимо сначала загрузить его с диска в память. Это лишний шаг, который не понадобится при использовании запроса POST или RPC.

Скорость загрузки сервиса хранения файлов ограничивает частоту запросов на генерацию миниатюр. Если сервис хранения файлов загружает файлы быстрее, чем сервис генерации миниатюр может генерировать и загружать миниатюры, топик Kafka предотвращает перегрузку запросами сервиса генерации миниатюр.

Решение с оркестрованной сагой

Отправку файлов и процесс генерации миниатюр также можно реализовать в виде оркестрованной саги. Сервис бэкенда является оркестратором. На рис. 12.6 представлены основные этапы генерации миниатюр с помощью оркестрованной саги:

1. Первый этап такой же, как для хореографического решения. Клиент отправляет запрос GET к бэкенду, чтобы проверить, было ли изображение уже загружено.
2. Сервис бэкенда загружает файл в сервис хранения объектов (не показан на рис. 12.6) через сервис хранения файлов. Сервис хранения файлов отправляет событие в топик ответов хранилища файлов, сообщая об успешной загрузке.
3. Сервис бэкенда потребляет событие из топика ответов хранилища файлов.
4. Сервис бэкенда отправляет событие в топик CDN для запроса файла, выгружаемого в CDN.
5. (a) Сервис хранения файлов потребляет из топика CDN и (b) загружает файл в CDN. Это делается на этапе, отдельном от загрузки данных в сервис хранения объектов, так что если загрузка в CDN завершится неудачей, повторение этого этапа не потребует повторной загрузки в сервис хранения объектов. В схеме, которая точнее соответствует подходу оркестрации, сервис бэкенда получает файл из сервиса хранения файлов, после чего загружает его в CDN. Вы принимаете решение: либо придерживаться принципа оркестрации от начала до конца, либо временно отклониться от него, чтобы файл не приходилось перемещать между тремя сервисами. Следует учитывать, что если вы выберете вариант с отклонением, вам придется настроить сервис хранения файлов на отправку запросов к CDN.
6. Сервис хранения файлов отправляет событие в топик ответов CDN, сообщая о том, что файл успешно загружен в CDN.
7. Сервис бэкенда потребляет из топика ответов CDN.
8. Сервис бэкенда отправляет событие в топик генерации миниатюр, с указанием сгенерировать миниатюры выгруженного изображения.
9. Сервис генерации миниатюр потребляет из топика генерации миниатюр.
10. Сервис генерации миниатюр получает файл от сервиса хранения файлов, генерирует миниатюры и записывает их в сервис хранения файлов.

11. Сервис генерации миниатюр отправляет событие в топик хранения файлов, сообщая, что генерация миниатюр прошла успешно.
12. Сервис хранения файлов потребляет событие из топика хранения файлов и отправляет миниатюры в CDN. Здесь применимо то, что говорилось об оркестрации и сетевом трафике в описании этапа 4.

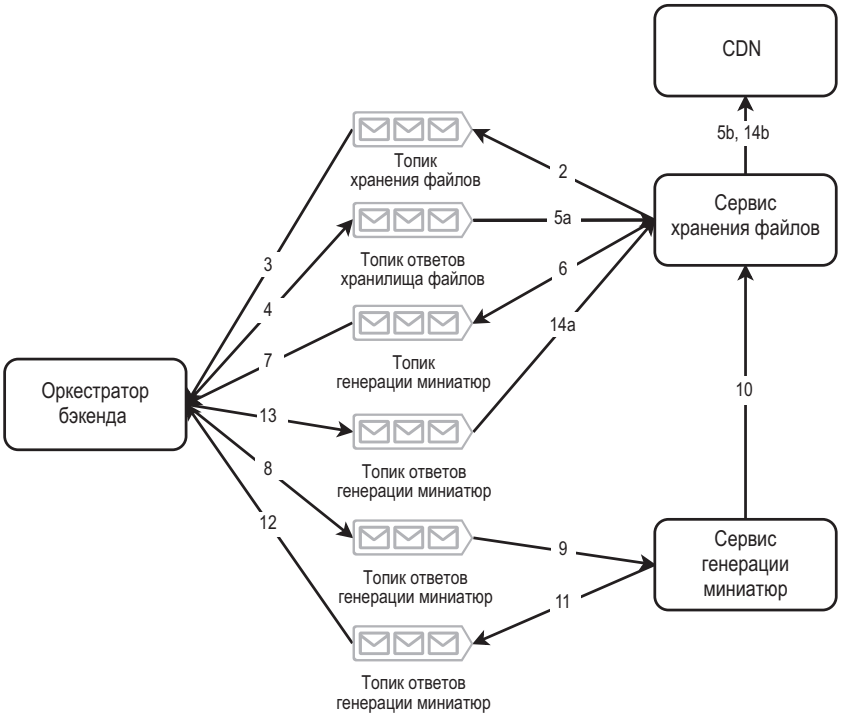

Рис. 12.6. Оркестрация генерации миниатюр, начиная с этапа 2. На рис. 12.5 был изображен сервис хранения объектов, который для простоты не представлен на этой диаграмме. Здесь также не представлен пользователь

12.6.3. Реализация генерации на стороне сервера и на стороне клиента

Генерирация может быть реализована сразу на стороне сервера и на стороне клиента. Можно начать с реализации генерации на стороне сервера, чтобы иметь возможность генерировать миниатюры для любого клиента. Затем реализуется генерация на стороне клиента для каждого типа клиента, чтобы воспользоваться преимуществами генерации на стороне клиента. Сначала миниатюру генерирует клиент, а в случае неудачи миниатюра будет сгенерирована сервером. С таким подходом исходным реализациям генерации на стороне клиента не нужно

учитывать все возможные сценарии отказов, и можно проводить улучшение генерации на стороне клиента пошагово.

Этот способ более сложный и затратный, чем простая генерация на стороне сервера, но он может быть проще и дешевле даже простой генерации на стороне клиента, потому что генерация на стороне сервера остается резервным механизмом на случай отказа, так что ошибки и сбои на стороне клиента будут обходиться дешевле. Клиентам можно назначать коды версий, которые будут включаться в запросы. Располагая информацией об ошибках в конкретной версии, можно настроить генерацию на стороне сервера так, чтобы она выполнялась для всех запросов, отправляемых такими клиентами. Затем вы исправляете ошибки, предоставляете новую версию клиента и оповещаете соответствующих пользователей о необходимости обновления клиентов. Даже если некоторые пользователи не выполнят обновление, это не станет серьезной проблемой, потому что такие операции могут выполняться на стороне сервера, а клиентские устройства будут постепенно устаревать и выходить из употребления.

12.7. СОХРАНЕНИЕ (ВЫГРУЗКА) ИЗОБРАЖЕНИЙ И ДАННЫХ

Изображения и миниатюры загружены в CDN и готовы к просмотру. Запрос, отправляемый зрителем на получение миниатюр автора, обрабатывается по следующей схеме.

1. Запрос к таблице Share для получения списка авторов, которые позволяют зрителям просматривать свои изображения.
2. Запрос к таблице Image для получения всех значений CdnPath для изображений пользователя, имеющих разрешение миниатюр. Запрос возвращает значения CdnPath и временный токен OAuth2 для чтения из CDN.
3. Затем клиент сохраняет миниатюры из CDN. Чтобы гарантировать, что клиент авторизован для выгрузки запрашиваемых файлов, CDN может использовать механизм авторизации на базе токенов, который подробно описан в разделе 13.3.

Динамический контент может обновляться или удаляться, поэтому мы храним его в SQL, а не в CDN. К этой категории относятся комментарии к фотографиям, информация профилей пользователей и пользовательские настройки. Для популярных миниатюр и популярных изображений в полном разрешении используется кэш Redis. Когда зритель помечает изображение как избранное, мы можем воспользоваться преимуществами неизменяемости изображений и кэшировать как миниатюры, так и изображение в полном разрешении на стороне клиента, если там хватает свободного места. Тогда запрос пользователя

на просмотр сетки изображений не будет потреблять ресурсы сервера и будет обрабатываться мгновенно.

Если в контексте собеседования вам не разрешается использовать доступный сервис CDN, то данная задача преобразуется в задачу о том, как спроектировать CDN. Эта тема обсуждается в следующей главе.

12.7.1. Загрузка страниц с миниатюрами

Рассмотрим сценарий, в котором пользователь в каждый момент времени просматривает одну страницу миниатюр. Допустим, на одной странице размещаются 10 миниатюр — на странице 1 миниатюры 1–10, на странице 2 — миниатюры 11–20 и т. д. Если новая миниатюра (назовем ее миниатюра 0) появится, когда пользователь находится на странице 1, а затем пользователь переходит к странице 2, как убедиться, что запрос пользователя на просмотр страницы 2 вернет ответ с миниатюрами 11–20, а не 10–19?

Одно из возможных решений — версионирование разбивки на страницы вида `GET thumbnails?page=<`*страница*`>&page_version=<`*версия_страницы*`>`. Если *версия_страницы* отсутствует, бэкенд подставляет новейшую версию по умолчанию. Ответ на запрос должен содержать *версию_страницы*, чтобы пользователь мог продолжить пользоваться тем же значением для последующих запросов. В этом случае переход по страницам будет плавным. Когда пользователь возвращается к странице 1, он может опустить версию страницы, и будет выведена последняя версия страницы 1 с миниатюрами.

Впрочем, этот прием работает только при добавлении или удалении миниатюр в начало списка. Если они добавляются в других местах списка, когда пользователь листает страницы, он не увидит новые миниатюры или продолжит видеть удаленные миниатюры. Более удачное решение — передача текущего первого или последнего элемента на бэкенд. Если пользователь листает страницы в направлении от первой к последней, используйте `GET thumbnails?previous_last=<`*посл_элемент*`>`. Если пользователь листает страницы в обратном направлении, используйте `GET thumbnails?previous_irst=<`*первый_элемент*`>`. Читатель может сам разобраться, как работает это решение.

12.8. МОНИТОРИНГ И ОПОВЕЩЕНИЯ

Помимо того, что обсуждалось в разделе 2.5, следует организовать мониторинг и оповещения как для загрузки, так и для сохранения файлов, а также для запросов к базе данных SQL.

12.9. ДРУГИЕ СЕРВИСЫ

Существует много других сервисов, которые можно обсудить в любом порядке: монетизация (реклама, премиум-функциональность), механизмы оплаты, цензура, персонализация и т. д.

12.9.1. Премиум-функциональность

Сервис фотохостинга может предлагать бесплатный тариф со всей функциональностью, обсуждавшейся ранее. Кроме нее, может существовать платная расширенная функциональность. Например, авторы могут указать, что фотографии защищены авторским правом, и зрители должны заплатить, чтобы сохранить версию фотографии в полном разрешении и использовать ее. Систему можно спроектировать так, чтобы авторы могли продавать свои фотографии другим пользователям. Для этого необходимо хранить значение объема продаж и отслеживать владельцев фотографий. Возможно, продавцам будут предоставляться метрики продаж, дашборды и аналитические средства для принятия более эффективных бизнес-решений. Также можно реализовать системы рекомендаций для продавцов, сообщающие, какие фотографии могут пользоваться спросом, и предлагающие цену. Все эти средства могут быть бесплатными или платными.

Для бесплатных учетных записей можно установить лимит в 1000 фотографий, а для разных планов подписки — более высокий порог. Возможно, для премиум-функциональности также стоит спроектировать сервисы использования и тарификации.

12.9.2. Сервисы платежей и налогов

Премиум-функциональность требует сервисов платежей и налогов для управления транзакциями и переводом средств между пользователями. Как отмечается в разделе 15.1, платежи — очень сложная тема, по которой во время собеседований по проектированию систем вопросов обычно не задают. Эксперт может задать их для дополнительной проверки знаний. Все сказанное относится и к налогам. Существует много разновидностей налогов: налог с продаж, подоходный налог, корпоративный налог и т. д. Каждая разновидность может включать несколько компонентов: налоги страны, штата, округа, города и т. д. Могут существовать правила освобождения от налогов в зависимости от уровня дохода, конкретного продукта, отрасли и т. д. Ставка налога может быть фиксированной или прогрессивной. Возможно, вам придется предусмотреть соответствующие формы налога на прибыль или подоходного налога для мест, в которых были сделаны и проданы фотографии.

12.9.3. Цензура/модерирование контента

Цензура, также часто называемая модерированием контента, играет важную роль в каждом приложении, в котором пользователи обмениваются данными друг с другом. Ваша этическая (а во многих случаях и юридическая) обязанность — контролировать приложение и удалять недопустимый или оскорбительный контент независимо от того, является ли он общедоступным или передается только между определенными пользователями.

Вам придется спроектировать систему для модерирования контента. Модерирование можно выполнять как вручную, так и автоматически. К ручным методам относятся механизмы, позволяющие зрителям сообщать о неуместном контенте, а группе эксплуатации — просматривать и удалять этот контент. Для модерирования контента можно применять эвристику и методы машинного обучения. Система должна предоставлять такие административные средства, как система предупреждений или блокировки авторов, и упростить группе эксплуатации взаимодействие с правоохранительными органами.

12.9.4. Реклама

Клиенты могут показывать пользователям рекламу. Самый частый способ — использование стороннего рекламного SDK, предоставляемого рекламной сетью (например, Google Ads). Рекламная сеть предоставляет распространителю рекламы (то есть вам) консоль для выбора категорий, которые вы хотите (или не хотите) выводить в клиенте. Например, можно запретить показ рекламы 18+ или рекламы компаний-конкурентов.

Другая возможность — проектирование внутренней системы для вывода рекламы в клиенте. Авторы могут выводить рекламу в клиенте, чтобы повысить уровень продаж своих фотографий. Один из практических сценариев — поиск фотографий, которые покупают зрители для своих целей. На домашней странице приложения могут выводиться рекомендованные зрителю фотографии; автор может заплатить, чтобы его фотографии появились в их числе. Также при поиске фотографий можно выводить «спонсируемые» результаты поиска.

Можно предоставить пользователю пакеты платной подписки в обмен на показ контента без рекламы.

12.9.5. Персонализация

Когда сервис начнет масштабироваться для большого количества пользователей, стоит подумать о персонализации контента, чтобы сделать его привлекательным для более широкой аудитории и повысить доход. В зависимости от активности пользователя в приложении и пользовательских данных, полученных из других источников, можно предоставить пользователю персонализированную рекламу, поиск и рекомендации контента.

Теория обработки данных и алгоритмы машинного обучения обычно выходят за рамки собеседований по проектированию систем, а обсуждение ведется вокруг проектирования экспериментальных платформ для разделения пользователей на группы, назначения каждой группе отдельной модели машинного обучения, сбора и анализа результатов и масштабирования успешных моделей на более широкую аудиторию.

12.10. ДРУГИЕ ВОЗМОЖНЫЕ ТЕМЫ ОБСУЖДЕНИЯ

Другие возможные темы обсуждения:

- Можно создать индекс Elasticsearch для метаданных фотографий: названия, описания и тегов. При отправке пользователем поискового запроса проводится нечеткое сопоставление тегов, а также названий и описаний. Обсуждение кластеров Elasticsearch см. в разделе 2.6.3.
- Ранее упоминалось о том, как авторы могут предоставлять зрителям доступ к своим изображениям. Можно более подробно обсудить средства управления доступом к изображениям, например управление доступом на уровне отдельных изображений, права на сохранение изображений в разных разрешениях или права на отправку изображений ограниченному количеству других зрителей. Также можно обсудить управление доступом к профилям пользователей. Пользователь может либо разрешить просмотр профиля всем желающим, либо предоставлять доступ каждому по отдельности. Приватные профили должны быть исключены из результатов поиска.
- Также можно обсудить другие способы организации фотографий. Например, авторы могут включать фотографии в группы. Группы могут включать фотографии разных авторов. Чтобы пользователь мог просматривать и/или публиковать фото в группе, он должен быть участником этой группы. Группе могут назначаться пользователи-администраторы, которые могут добавлять и удалять пользователей из группы. Вы можете обсудить разные способы упаковки и продажи коллекций фотографий, а также соответствующие варианты дизайна системы.
- Еще одна возможная тема для обсуждения — управление авторскими правами и «водяные знаки». Пользователь может закрепить за каждой фотографией конкретный вариант авторской лицензии. Система может добавлять на фотографию невидимый «водяной знак», а также дополнительные «водяные знаки» во время транзакций между пользователями. «Водяные знаки» могут использоваться для отслеживания прав владельца и обнаружения нарушений авторских прав.
- Пользовательские данные (файлы изображений) в системе требуют особого обращения и обладают ценностью. Можно обсудить риск потери данных, ее

предотвращение и способы сокращения последствий. К этой категории также относятся бреши в безопасности и хищение данных.

- Можно обсудить стратегии контроля затрат на хранение данных. Например, использование разных систем хранения данных для новых и старых файлов или для популярных и прочих изображений.
- Можно создать пакетные пайплайны для аналитики, например пайплайн для определения самых популярных фотографий или подсчета выгруженных фотографий по часам, дням и месяцам. Такие пайплайны рассматриваются в главе 17.
- Пользователь может подписаться на публикации другого пользователя, чтобы получать оповещения о новых фотографиях и/или комментариях.
- Систему можно расширить для поддержки потоковой передачи аудио и видео. Для обсуждения потокового видео обязательно знание предметной области, которое не требуется на общих собеседованиях по проектированию систем. Эта тема может подниматься на собеседованиях на вакансии, для которых требуется это знание, или как дополнительный вопрос для проверки эрудиции соискателя.

ИТОГИ

- Масштабируемость, доступность и высокая скорость выгрузки (download), то есть сохранения файлов, — необходимые требования для сервиса фотохостинга или хостинга файлов. Высокая скорость загрузки (upload) и консистентность не обязательны.
- Каким сервисам разрешается запись в CDN? Используйте CDN для статических данных, но защищайте и ограничивайте доступ для записи к чувствительным сервисам (таким, как CDN).
- Какие операции обработки должны выполняться на стороне клиента или сервера? Один из факторов, который следует учитывать: обработка на стороне клиента экономит аппаратные ресурсы и средства, но существенно повышает сложность, а это влечет дополнительные затраты.
- У обработки на стороне клиента и на стороне сервера есть свои достоинства и недостатки. Обработка на стороне сервера обычно предпочтительна там, где возможно, так как она упрощает разработку/обновления. Выполнение обработки на обеих сторонах позволяет совместить низкие вычислительные затраты (клиент) с высокой надежностью (сервер).
- Какие процессы могут быть асинхронными? Используйте для них такие средства, как саги, в целях улучшения масштабируемости и сокращения затрат на оборудование.

13 Проектирование CDN

В ЭТОЙ ГЛАВЕ

- ✓ Достоинства, недостатки и непредвиденные ситуации
- ✓ Обслуживание пользовательских запросов с архитектурой хранения метаданных на фронтенде
- ✓ Проектирование базовой системы распределенного хранения данных

CDN (Content Distribution Network, сеть распространения контента) — эффективный, географически распределенный сервис хранения файлов. CDN предназначается для репликации файлов по нескольким датацентрам, чтобы быстро предоставлять статический контент большому количеству географически распределенных пользователей. Каждый пользователь обслуживается тем датацентром, который может делать это быстрее других. Среди других преимуществ CDN — отказоустойчивость, позволяющая обслуживать пользователей из других датацентров, если конкретный датацентр станет недоступным. Рассмотрим возможный дизайн сети CDN, которую назовем CDNService.

13.1. ДОСТОИНСТВА И НЕДОСТАТКИ CDN

Прежде чем обсуждать требования и системный дизайн CDN, рассмотрим достоинства и недостатки CDN. Это поможет лучше понять суть требований.

13.1.1. Достоинства CDN

Если компания размещает сервисы в нескольких датацентрах, скорее всего, понадобится общее хранилище объектов, реплицируемое между этими датацентрами для обеспечения избыточности и доступности. Общее хранилище объектов обеспечивает многие преимущества CDN. Обычно мы используем CDN, если обширная сеть датацентров, предоставляемая CDN, принесет пользу географически распределенной пользовательской базе.

Обоснования для использования CDN обсуждались в разделе 1.4.4. Напомним некоторые из них:

- *Снижение задержки* — пользователь обслуживается ближайшим датацентром, что снижает задержку. Без сторонней CDN пришлось бы разворачивать сервис на нескольких датацентрах, что добавляет существенную сложность, например мониторинг для обеспечения доступности. Снижение задержки может иметь и другие преимущества, например улучшение показателей SEO (поисковой оптимизации). Поисковые системы обычно «штрафуют» медленные веб-страницы как прямо, так и косвенно. Примером косвенного штрафа можно назвать ситуацию, когда пользователи покидают медленно загружающийся веб-сайт; говорят, что у такого сайта высокий показатель отказов (bounce rate). Поисковые системы «штрафуют» сайты с высокими показателями отказов.
- *Масштабируемость* — со сторонним провайдером не нужно заниматься масштабированием системы самостоятельно. Провайдер берет масштабирование на себя.
- *Снижение удельной стоимости* — сторонние CDN обычно предоставляют скидки за объем, так что при обслуживании большего количества пользователей и более высоких нагрузках удельная стоимость снижается. CDN может обеспечить снижение затрат благодаря экономии на масштабе от обслуживания трафика многих компаний и распределению затрат на оборудование и квалифицированный технический персонал по большему объему. Колебания аппаратных и сетевых требований нескольких компаний компенсируют друг друга, что приводит к более стабильному спросу по сравнению с обслуживанием всего одной компании.
- *Повышение пропускной способности* — CDN предоставляет сервису дополнительные хосты, что позволяет обслуживать большее количество одновременно обслуживаемых пользователей и более высокий трафик.
- *Повышение доступности* — дополнительные хосты могут послужить резервным ресурсом на случай сбоев хостов сервиса, особенно если CDN удается соблюдать условия SLA. Географическое распределение по многим датацентрам также полезно для доступности, так как сбой одного датацентра не повлияет на другие датацентры, находящиеся на большом расстоянии. Более

того, непредвиденные выбросы трафика в одном датацентре могут быть перенаправлены и сбалансированы между другими датацентрами.

13.1.2. Недостатки CDN

Достоинства CDN широко известны, но мало кто упоминает о недостатках. Признаком зрелости специалиста является его умение обсуждать и оценивать сильные и слабые стороны любого технического решения и предвидеть возражения, которые могут возникнуть у других инженеров. Эксперт почти всегда будет оспаривать ваши проектировочные решения и проверять, учли ли вы разные нефункциональные требования. Вот некоторые недостатки использования CDN:

- Дополнительная сложность, обусловленная включением еще одного сервиса в систему. Примеры такой сложности:
  - Дополнительный поиск в DNS.
  - Дополнительная потенциальная точка отказа.
- CDN может дорого обходиться при низком трафике. Также могут присутствовать скрытые затраты, скажем, тарифы на передачу гигабайта данных, потому что CDN могут использовать сторонние сети.
- Миграция на другую сеть CDN обычно занимает месяцы и требует значительных затрат. Причины для миграции на другую CDN:
  - У конкретной CDN может не быть хостов, расположенных рядом с вашими пользователями. Если вы приобрели значительную пользовательскую базу в регионе, который не покрывается вашей CDN, возможно, вам придется мигрировать на более подходящую CDN.
  - Компания-провайдер CDN может прекратить свою деятельность.
  - Компания-провайдер CDN может не обеспечить надлежащего качества сервиса, например нарушать условия SLA, что влияет на пользователей; предоставлять плохую техническую поддержку; допускать такие инциденты, как потеря данных или дефекты безопасности.
- Некоторые страны или организации могут блокировать IP-адреса определенных CDN.
- Хранение данных у третьих сторон может быть сопряжено с проблемами безопасности и конфиденциальности. Вы можете реализовать шифрование, чтобы CDN не могла просматривать данные, но это повлечет дополнительные расходы и задержку (на шифрование и дешифрование данных). Над дизайном и реализацией должны работать квалифицированные инженеры в области безопасности, что добавит лишние финансовые затраты и усложнит коммуникацию.
- Другая возможная проблема безопасности — риск вставки вредоносного кода в библиотеки JavaScript. Вы не сможете лично удостовериться в безопасности и целостности этих библиотек, размещенных на удаленных хостах.

- Недостаток обеспечения высокой доступности третьей стороной: если в CDN возникнут технические проблемы, количество времени, которое потребуется компании CDN для их исправления, неизвестно. Любое снижение качества сервиса может повлиять на потребителей, а затраты на коммуникации с внешней компанией могут превысить затраты на коммуникации внутри компании. Компания CDN может предоставить соглашение SLA, но нельзя гарантировать, что оно будет соблюдаться, а миграция на другую CDN обойдется дорого, о чем говорилось выше. Более того, соглашение SLA начинает зависеть от третьей стороны.
- Управление конфигурацией CDN или любого стороннего инструмента/сервиса в целом может не обеспечивать необходимых настроек для некоторых практических сценариев, что может привести к непредвиденным проблемам. Пример рассматривается в следующем разделе.

13.1.3. Пример непредвиденной проблемы, когда CDN используется для поставки изображений

В этом разделе рассматривается пример непредвиденной проблемы, которая может возникнуть при использовании как CDN, так и сторонних инструментов или сервисов вообще.

CDN может прочитать заголовок User-Agent (https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/User-Agent) запроса GET, чтобы определить, поступил ли запрос от веб-браузера. Если проверка дает положительный результат, изображения возвращаются в формате WebP (https://developers.google.com/speed/webp) вместо формата, в котором они были выгружены (например, PNG или JPEG).

Для некоторых сервисов такое решение подходит идеально, но для браузерных приложений, требующих возврата изображений в исходном формате, возможны три варианта:

1. Переопределить заголовок User-Agent в веб-приложении.
2. Настроить CDN так, чтобы части сервисов предоставлялись изображения WebP, а остальным — изображения в исходных форматах.
3. Маршрутизировать запрос через сервис бэкенда.

Что касается решения 1, на момент публикации книги браузер Chrome не позволял приложениям переопределять заголовок User-Agent, тогда как Firefox давал такую возможность (см. https://bugs.chromium.org/p/chromium/issues/detail?id=571722, https://bugzilla.mozilla.org/show_bug.cgi?id=1188932, и https://stackoverlow.com/a/42815264). Таким образом, решение 1 ограничивает пользователей конкретными веб-браузерами, что может быть неприемлемо.

Теперь о решении 2: в CDN может отсутствовать возможность подобной настройки для отдельных сервисов. Может оказаться, что CDN позволяет только

включить или отключить предоставление изображений в формате WebP для всех сервисов. Но даже если индивидуальная настройка поддерживается, инфраструктурная группа вашей компании, управляющая конфигурациями CDN, может не захотеть (или не иметь возможности) настраивать эту конфигурацию для отдельных сервисов. Эта проблема более актуальна для крупных компаний.

Решение 3 требует предоставления конечной точки API только для загрузки исходного изображения из CDN. Этого решения лучше избегать, потому что с ним теряются многие преимущества CDN. Оно добавляет задержку и сложность (включая затраты на документирование и обслуживание). Хост бэкенда может быть географически отдален от пользователя, так что пользователь лишается преимущества обслуживания из ближайшего датацентра. Этот сервис бэкенда должен будет масштабироваться в зависимости от потребности в изображениях; если частота запросов на получение изображений высока, и CDN и сервис бэкенда необходимо масштабировать. Вместо того чтобы принимать это решение, разумнее хранить файлы в более дешевом хранилище объектов, хосты которого находятся в одном датацентре с сервисом бэкенда. К сожалению, я видел, как это «решение» используется в больших компаниях, потому что приложение и CDN принадлежали разным командам, а руководство не было заинтересовано в сотрудничестве между ними.

13.2. ТРЕБОВАНИЯ

Функциональные требования просты. Авторизованные пользователи должны иметь возможность создавать каталоги, отправлять файлы размером не более 10 Гбайт и сохранять (выгружать) файлы.

> **ПРИМЕЧАНИЕ** Тема модерирования контента обсуждаться не будет. Модерирование контента играет важную роль в приложении, в котором пользователи просматривают контент, созданный другими. Предполагается, что за модерирование отвечают организации, использующие ваш сервис CDN, а не компания, предоставляющая CDN.

Многие нефункциональные требования относятся к преимуществам CDN:

- *Масштабируемость* — CDN может масштабироваться для поддержки петабайт объема хранилища и загрузки терабайтных объемов в течение дня.
- *Высокая доступность* — требуется время простоя от четырех до пяти девяток.
- *Высокая производительность* — файл должен выгружаться из датацентра, который способен быстрее предоставить его источнику запроса. Однако синхронизация требует времени, так что производительность отправки не столь важна при условии, что до завершения синхронизации файл будет доступен хотя бы в одном датацентре.
- *Надежность* — файл не должен быть поврежден.

- *Безопасность и конфиденциальность* — CDN обслуживает запросы и отправляет файлы приемникам за пределами датацентра. Загрузка (upload) и выгрузка (download) файлов должны выполняться только авторизованными пользователями.

13.3. АУТЕНТИФИКАЦИЯ И АВТОРИЗАЦИЯ В CDN

Как показано в приложении Б, аутентификация проверяет, что пользователь является именно тем, за кого себя выдает, тогда как авторизация проверяет, что пользователь, обращающийся к ресурсу (например, файлу в CDN), имеет необходимые для этого разрешения. Эти проверки препятствуют *использованию прямых ссылок*, при котором сайт или сервис обращается к ресурсам CDN без разрешения. CDN теряет доход, предоставляя данные пользователям, которые за них не заплатили, а несанкционированные обращения к файлам или данным могут нарушать авторские права.

> **СОВЕТ** Подробнее ознакомиться с понятиями аутентификации и авторизации можно в приложении Б.

Процессы аутентификации и авторизации CDN могут базироваться либо на файлах cookie, либо на токенах. Как обсуждается в разделе Б.4, аутентификация на базе токенов расходует меньше памяти, может использовать более безопасные сторонние сервисы и поддерживать более детализированные средства управления доступом. Кроме этих преимуществ, аутентификация на базе токенов для CDN позволяет ограничить источники запросов разрешенными IP-адресами или конкретными учетными записями пользователей.

В этом разделе обсуждается типичная реализация аутентификации и авторизации в CDN. В следующих разделах рассматривается дизайн CDN, который можно обсудить в ходе собеседования, включая организацию процессов аутентификации и авторизации в системе.

13.3.1. Основные этапы аутентификации и авторизации в CDN

В этом обсуждении под потребителем CDN подразумевается сайт или сервис, который выгружает ресурсы в CDN, а затем направляет в CDN своих пользователей/клиентов. Под пользователем CDN понимается клиент, загружающий ресурсы из CDN.

CDN назначает каждому потребителю секретный ключ и предоставляет SDK или библиотеку для генерации токенов доступа из следующей информации. На рис. 13.1 показан процесс генерации токенов доступа.

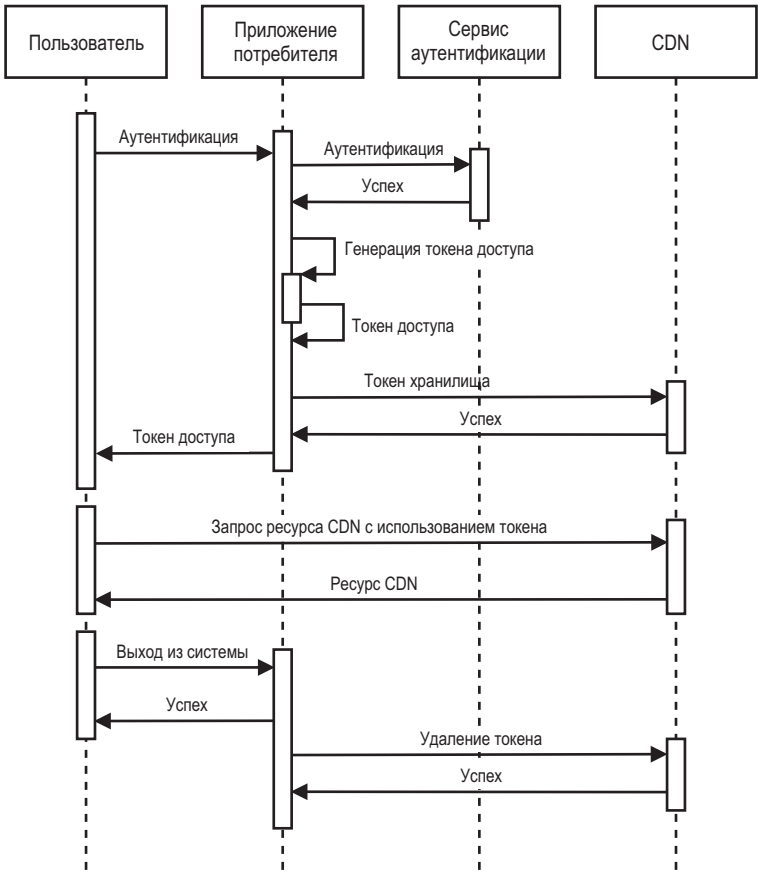

Рис. 13.1. Диаграмма последовательности действий процесса генерации токенов, последующего использования токена для запроса ресурсов CDN и уничтожения токена при выходе из системы. Процесс уничтожения токена может быть асинхронным, как показано на диаграмме, или синхронным, потому что выход из системы является относительно редким событием

1. Пользователь отправляет запрос аутентификации клиентскому приложению. Приложение потребителя может выполнять аутентификацию с использованием сервиса аутентификации. (Подробности механизма аутентификации не имеют отношения к процессу генерации токенов доступа CDN. Введение в различные протоколы аутентификации, такие как Simple Login и OpenID Connect, содержится в приложении Б.)
2. Приложение потребителя генерирует токен доступа с использованием SDK со следующими входными данными:

- *Секретный ключ* — секретный ключ потребителя.
- *CDN URL* — URL-адрес в CDN, для которого действителен сгенерированный токен доступа.
- *Срок действия* — метка времени срока действия токена доступа, по истечении которого пользователю понадобится новый токен доступа. Когда пользователь обращается с запросом к CDN, передавая токен с истекшим сроком, CDN возвращает ответ 302, чтобы перенаправить пользователя к потребителю. Потребитель генерирует новый токен доступа, а затем возвращает его пользователю с ответом 302 для повторения попытки запроса к CDN.
- *Referrer* — заголовок Referrer запроса HTTP.

Заголовок Referrer и безопасность

Когда клиент/пользователь отправляет запрос HTTP к CDN, он должен включить URL клиента в заголовке HTTP Referrer. CDN допускает только авторизованные источники ссылок, поэтому такая мера предотвращает использование CDN неавторизованными источниками.

Однако это не полноценный механизм безопасности. Клиенты могут легко фальсифицировать заголовки Referrer, просто используя в них другой URL. Сайт/сервис может выдать себя за авторизованный, и клиенты будут полагать, что взаимодействуют с авторизованным сайтом/сервисом.

- *Разрешенные IP-адреса* — может существовать список диапазонов IP-адресов, которым разрешена загрузка ресурсов CDN.
- *Разрешенные страны или регионы* — можно включить черный или белый список стран/регионов. Поле «разрешенные IP-адреса» уже указывает, какие страны/регионы разрешены, но можно добавить и это поле для удобства.

3. Приложение потребителя сохраняет токен, после чего возвращает его пользователю. Для дополнительной безопасности токен может храниться в зашифрованной форме.
4. Каждый раз, когда приложение потребителя предоставляет пользователю URL в CDN и пользователь отправляет запрос GET к этому ресурсу CDN, запрос GET должен быть *подписан* токеном доступа. Пример подписанного URL: http://12345.r.cdnsun.net/photo.jpeg?secure=DMF1ucDxtHCxwYQ (из https://cdnsun.com/knowledgebase/cdn-static/setting-a-url-signing-protect-your-cdn-content). Здесь «secure=DMF1ucDxtHCxwYQ» — параметр запроса для отправки токена доступа в CDN. CDN выполняет авторизацию. При этом CDN проверяет, что токен пользователя действителен и ресурс может быть загружен

с этим токеном, а также проверяет IP-адрес или страну/регион пользователя. Наконец, CDN возвращает ресурс пользователю.

5. При выходе пользователя из системы приложение потребителя уничтожает его токен. При повторном входе пользователь должен будет сгенерировать другой токен.

Удаление токена может быть асинхронным, как на рис. 13.1, или же синхронным, потому что выход из системы является относительно редким событием. Если удаление токена выполняется асинхронно, возникает риск того, что токены не будут удалены, если на хосте приложения потребителя, обрабатывающего это удаление, произойдет неожиданный сбой. Одно из возможных решений — попросту проигнорировать эту проблему и допустить, что некоторые токены не будут уничтожены. Другое решение основано на использовании событийных механизмов. Хост приложения потребителя может отправлять события удаления токена в очередь Kafka, а другой кластер может потреблять эти события и удалять токены в CDN. В третьем решении удаление токена реализуется как синхронная/блокирующая операция. Если удаление токена завершится неудачей из-за непредвиденного сбоя на хосте приложения потребителя, пользователь/клиент получит ошибку 500 и клиент сможет повторно выполнить запрос на удаление. Такое решение приводит к более высокой задержке для запроса на выход из системы, но это может быть приемлемо.

За дополнительной информацией об аутентификации и авторизации на базе токенов в CDN обращайтесь к статьям https://docs.microsoft.com/en-us/azure/cdn/cdn-token-auth, https://cloud.ibm.com/docs/CDN?topic=CDN-working-with-token-authentication, https://blog.cdnsun.com/protecting-cdn-content-with-token-authentication-and-url-signing.

13.3.2. Ротация ключей

Ключ потребителя может периодически изменяться на случай его перехвата взломщиком. В этом случае ущерб будет ограничен, так как ключ будет полезен только до момента его смены.

Вместо внезапной замены ключа применяется механизм *ротации*. В ходе ротации присутствует период, в котором действительны как старые, так и новые ключи. Распространение нового ключа по всем системам потребителей занимает время, так что потребитель может продолжать пользоваться как старым, так и новым ключом. В заданное время старый ключ перестает действовать, и пользователи не могут обращаться к ресурсам CDN по ключам с истекшим сроком действия.

Также полезно установить эту процедуру для случаев, когда известно о похищении ключа взломщиком. CDN может провести ротацию и установить короткое время действия старого ключа. Потребитель переходит на новый ключ как можно быстрее.

13.4. ВЫСОКОУРОВНЕВАЯ АРХИТЕКТУРА

На рис. 13.2 изображена высокоуровневая архитектура CDN. В ней используется типичная архитектура «шлюз API — метаданные — хранилище/база данных». Запрос пользователя обрабатывается шлюзом API — уровнем/сервисом, который отправляет запросы к другим сервисам. (Обзор шлюзов API представлен в разделе 1.4.6.) Запросы включают завершение SSL, аутентификацию и авторизацию, ограничение частоты запросов (см. главу 8) и использование общего сервиса ведения журналов для таких целей, как аналитика и выставление счетов. Шлюз API можно настроить так, чтобы он при помощи сервиса метаданных определял, на каком хосте сервиса хранения должны осуществляться чтение и запись для каждого пользователя. Если ресурс CDN шифруется при хранении, сервис метаданных также может зарегистрировать этот факт, а вы можете воспользоваться сервисом управления секретами для управления ключами шифрования.

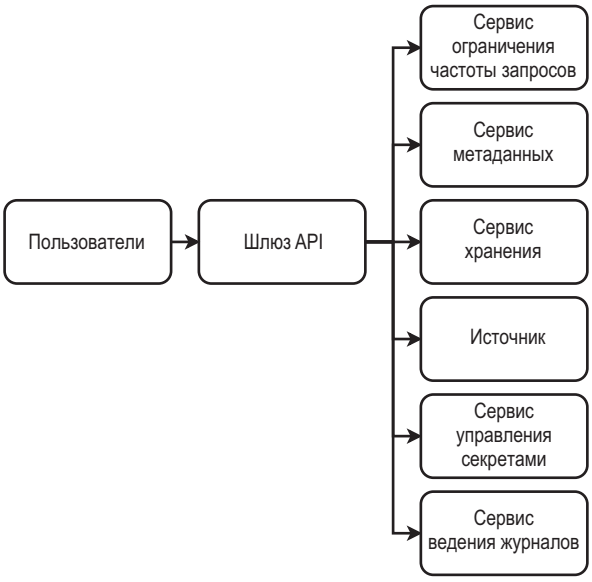

Рис. 13.2. Высокоуровневая архитектура CDN. Запросы пользователей маршрутизируются через шлюз API, который отправляет запросы к соответствующим сервисам, включая ограничение частоты запросов и ведение журналов. Ресурсы хранятся в сервисе хранения, и сервис метаданных отслеживает хосты сервиса хранения и каталоги с файлами, в которых хранятся ресурсы. Если ресурсы зашифрованы, для управления ключами шифрования используется сервис управления секретами. Если запрошенный ресурс отсутствует, шлюз API получает его от источника (то есть нашего сервиса; это настраивается в сервисе метаданных), добавляет его в сервис хранения и обновляет сервис метаданных

Операции можно обобщить по категориям чтения (загрузка) и записи (создание каталогов, отправка и удаление файлов). Для упрощения исходной архитектуры

каждый файл может быть реплицирован по всем датацентрам. В противном случае системе придется обрабатывать дополнительную сложность:

- Сервис метаданных должен будет отслеживать, в каких датацентрах хранятся те или иные файлы.
- Потребуется система распределения файлов, периодически использующая метаданные запросов пользователей для определения оптимального распределения файлов между датацентрами (в частности, количество и местонахождение реплик).

13.5. СЕРВИС ХРАНЕНИЯ

Сервис хранения представляет собой кластер хостов/узлов, содержащих файлы. Как отмечалось в разделе 4.2, для хранения больших файлов не следует использовать базы данных. Лучше хранить их в файловых системах хостов. Файлы должны реплицироваться в целях доступности и надежности, при этом каждый файл может быть закреплен за несколькими (скажем, тремя) хостами. Необходимо организовать мониторинг доступности и процесс отработки отказов, который обновляет сервис метаданных и предоставляет заменяющие узлы. Менеджер хоста может быть как внутрикластерным, так и внекластерным. Внутрикластерный менеджер напрямую управляет узлами, тогда как внекластерный менеджер управляет небольшими независимыми кластерами узлов, а каждый малый кластер управляет собой.

13.5.1. Внутрикластерный менеджер

Можно воспользоваться распределенной файловой системой, такой как HDFS, которая включает ZooKeeper в качестве внутрикластерного менеджера. ZooKeeper управляет выборами лидера и поддерживает связи между файлами, лидерами и последователями. Внутрикластерный менеджер — сложный компонент, который также должен быть надежным, масштабируемым и высокопроизводительным. Если такой компонент вам не требуется, существует альтернатива — внекластерный менеджер.

13.5.2. Внекластерный менеджер

Каждый кластер, находящийся под управлением внекластерного менеджера, состоит из трех и более узлов, распределенных по нескольким датацентрам. Чтобы прочитать или записать файл, сервис метаданных идентифицирует кластер, в котором он хранится или должен храниться, после чего читает или записывает файл из случайно выбранного узла в кластере. Этот узел отвечает за репликацию по другим узлам в кластере. Механизм выбора лидера не нужен, но сопоставление файлов с кластерами обязательно. Внекластерный менеджер поддерживает сопоставление файлов с кластерами.

13.5.3. Оценка

На практике внекластерный менеджер не проще внутрикластерного. В табл. 13.1 сравниваются эти два решения.

Таблица 13.1. Сравнение внутрикластерного менеджера с внекластерным

Внутрикластерный менеджер	Внекластерный менеджер
Сервис метаданных не отправляет запросы к внутрикластерному менеджеру	Сервис метаданных отправляет запросы к внекластерному менеджеру
Управляет распределением файлов в рамках конкретных ролей в кластере	Управляет распределением файлов в кластере, но не по отдельным узлам
Должен знать о каждом узле в кластере	Может не знать о каждом отдельном узле, но должен знать о каждом кластере
Отслеживает контрольные сигналы от узлов	Отслеживает работоспособность каждого независимого кластера
Обрабатывает сбои хостов. Узлы могут удаляться, и в кластер могут добавляться новые узлы	Отслеживает загрузку каждого кластера и решает проблемы перегрузки кластера. Новые файлы не могут назначаться в кластеры, достигшие предела вместимости

13.6. ТИПИЧНЫЕ ОПЕРАЦИИ

Когда клиент отправляет запрос к домену сервиса CDN (например, cdnservice.flickr.com) вместо IP-адреса, GeoDNS (см. разделы 1.4.2 и 7.9) назначает IP-адрес ближайшего хоста, а балансировщик нагрузки направляет его на хост шлюза API. Как описано в разделе 6.2, шлюз API выполняет ряд операций, включая кэширование. Сервис фронтенда и связанный с ним сервис кэширования могут участвовать в кэшировании файлов, к которым часто обращаются пользователи.

13.6.1. Чтение: выгрузка

Для выгрузки файлов следующим шагом становится выбор хоста хранения для обслуживания этого запроса. Сервис метаданных помогает сделать этот выбор, поддерживая и предоставляя следующие метаданные. Он может использовать Redis и/или SQL.

- Хосты сервиса хранения, содержащие файлы. Некоторые или все хосты могут храниться в других датацентрах, так что эту информацию тоже необходимо хранить. Репликация файлов между хостами требует времени.
- Сервис метаданных каждого датацентра отслеживает текущую нагрузку его хостов. Загрузку хоста можно приближенно оценить по суммарному размеру файлов, которые на нем хранятся.

- Если потребуется оценить, сколько времени займет выгрузка файла с хоста, или различить одноименные файлы (но обычно это делается с использованием хешей MD5 или SHA).
- Принадлежность файлов и управление доступом.
- Состояние работоспособности хостов.

Процесс выгрузки

На рис. 13.3 изображена диаграмма последовательности действий шлюза API для выгрузки файла, если этот ресурс хранится в CDN. Мы опустили некоторые шаги (завершение SSL, аутентификация и авторизация, запись в журнал).

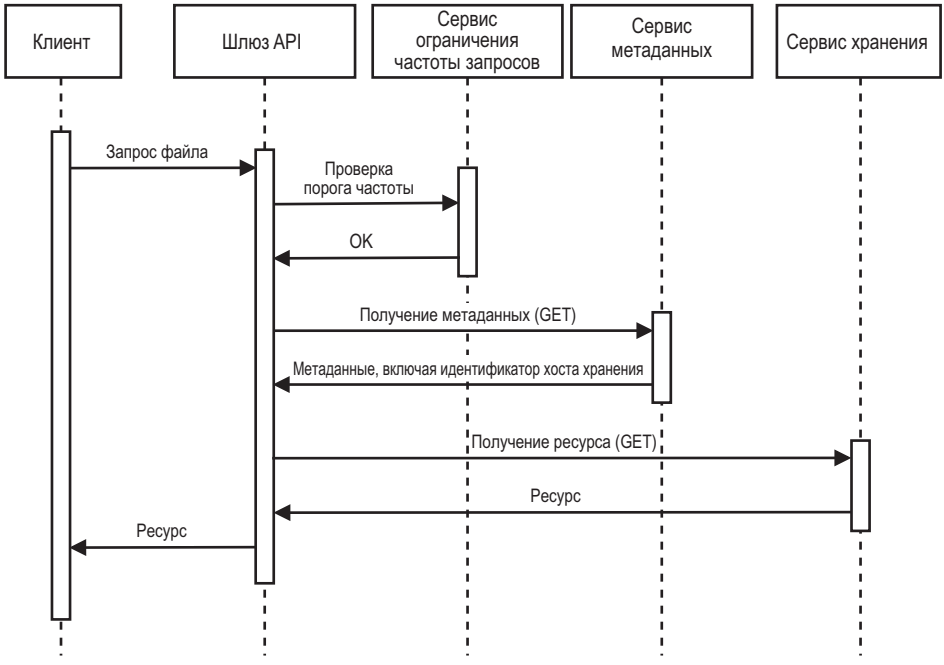

Рис. 13.3. Диаграмма последовательности действий клиента при выполнении загрузки из CDN. Предполагается, что ограничитель нагрузки пропустил запрос. Если ресурс присутствует, то последовательность действий прямолинейна

1. Запрос к сервису ограничения частоты для проверки того, не превысил ли запрос порог частоты клиента. Далее предполагается, что ограничитель частоты пропустил запрос.
2. Запрос к сервису метаданных для получения хостов сервиса хранения, содержащих этот ресурс.
3. Выбор хоста хранения и потоковая передача ресурса клиенту.

4. Обновление сервиса метаданных с повышением нагрузки хоста хранения. Если сервис метаданных хранит размер ресурса, этот шаг может выполняться параллельно с шагом 3. В противном случае шлюзу API придется измерить размер ресурса, чтобы обновить сервис метаданных верной величиной повышения нагрузки.

Внимательный читатель заметит, что последний шаг обновления нагрузки в сервисе метаданных шлюзом API может выполняться асинхронно. Если хост шлюза API столкнется со сбоем в ходе обновления, сервис метаданных может не получить обновление. Вы можете проигнорировать ошибку и разрешить пользователю использовать CDN в большей степени, чем ему разрешено. Возможен и другой вариант: хост шлюза API отправит это событие в топик Kafka. Потреблять события из этого топика может либо сервис метаданных, либо специально выделенный кластер потребления, который затем обновляет сервис метаданных.

Запрашиваемый ресурс может отсутствовать в CDN. Он мог быть удален по многим причинам, включая следующие:

- Для ресурсов может быть установлен период удержания данных, например, несколько месяцев или лет, и этот период истек для указанного ресурса. Период удержания также может быть основан на времени последнего обращения к ресурсу.
- Менее вероятная причина заключается в том, что ресурс так и не был отправлен, потому что в CDN кончилось свободное место (или произошли другие ошибки), но клиент полагал, что отправка прошла успешно.
- Другие ошибки в CDN.

На рис. 13.4 показано, что если ресурс отсутствует в CDN, его нужно будет выгрузить из источника — хранилища резервной копии, предоставленного клиентом. Это приведет к увеличению задержки. Выгруженный ресурс нужно будет сохранить, загрузив его в сервис хранилища и обновив сервис метаданных. Для минимизации задержки процесс хранения можно выполнять параллельно с возвращением ресурса клиенту.

Процесс загрузки с шифрованием при хранении

А если ресурсы должны храниться в зашифрованном виде? На рис. 13.5 можно хранить ключи шифрования в сервисе управления секретами (что требует аутентификации). При инициализации хост шлюза API осуществляет аутентификацию в сервисе управления секретами, который передаст хосту токен для будущих запросов. На рис. 13.5 при запросе ресурса авторизованным пользователем хост сначала получает ключ шифрования ресурса от сервиса управления секретами, загружает зашифрованный ресурс из сервиса хранилища, расшифровывает ресурс и возвращает его пользователю. Большой ресурс

может занимать несколько блоков в сервисе хранения, и каждый блок придется загружать и расшифровывать по отдельности.

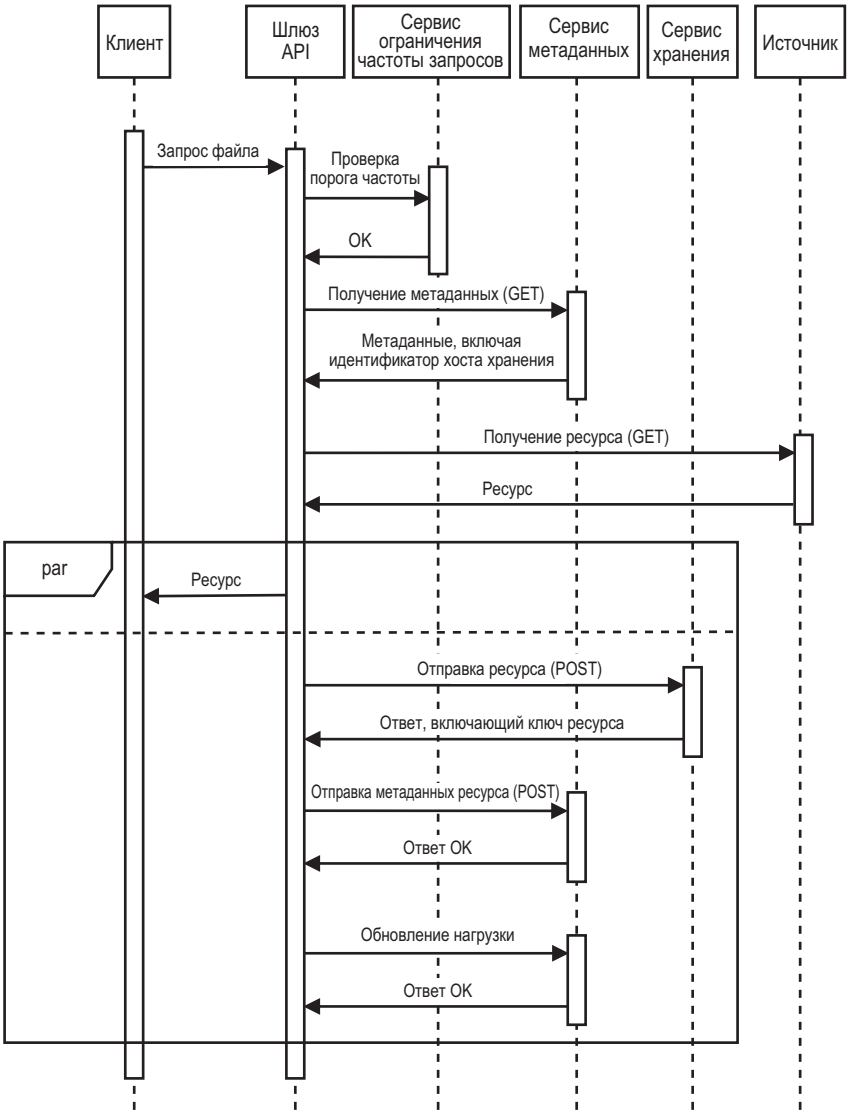

Рис. 13.4. Диаграмма последовательности действий при выгрузке из CDN, если CDN не содержит запрашиваемый ресурс. CDN необходимо выгрузить ресурс из источника (резервного хранилища, предоставляемого клиентом) и вернуть его пользователю, а также сохранить ресурс для будущих запросов. Отправка метаданных ресурса запросом POST и обновление нагрузки могут выполняться в одном запросе, но для простоты мы разделили их на разные

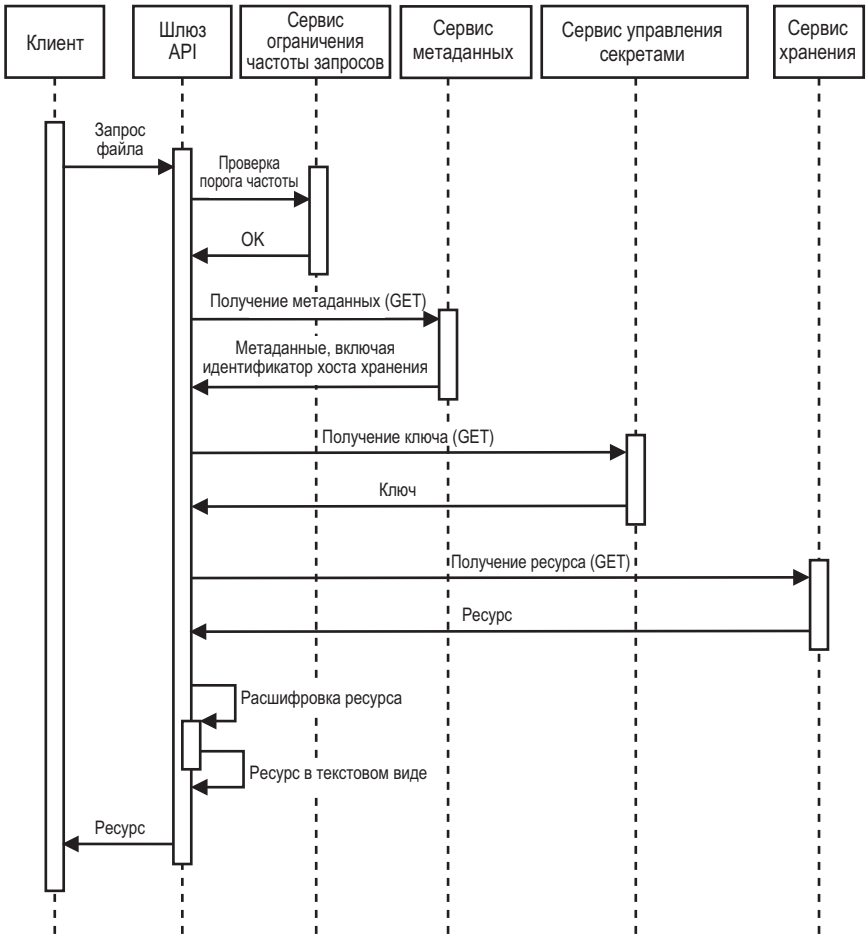

Рис. 13.5. Диаграмма последовательности действий выгрузки. Предполагается, что ресурс присутствует в CDN. Большой ресурс может занимать несколько блоков в сервисе хранения, и каждый блок придется выгружать и расшифровывать по отдельности

На рис. 13.6 показаны действия при отправке запроса на выборку зашифрованного ресурса, которым CDN не обладает. Как и в случае на рис. 13.5, шлюз API должен выгрузить ресурс из источника. Затем шлюз API возвращает ресурс пользователю параллельно с сохранением его в сервисе хранения. Шлюз API генерирует случайный ключ шифрования, зашифровывает ресурс, записывает ресурс в сервис хранения, а ключ — в сервис управления секретами.

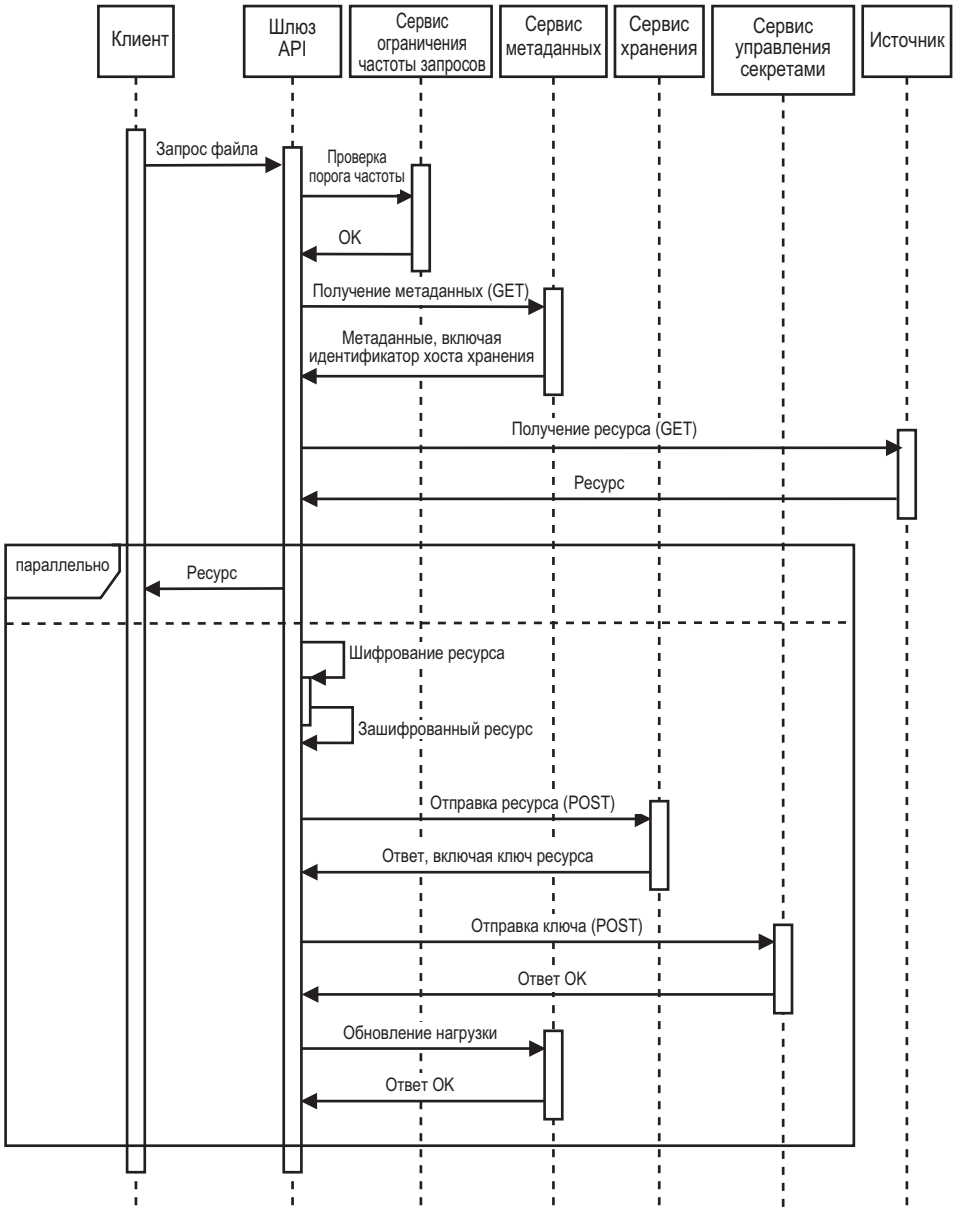

Рис. 13.6. Диаграмма последовательности действий при выгрузке зашифрованного файла. Запросы POST для ресурса и ключа могут выполняться параллельно

13.6.2. Запись: создание каталогов, отправка и удаление файлов

Файл определяется его идентификатором, а не содержимым. (Имена файлов не могут использоваться в качестве идентификатора, потому что разные пользователи могут давать одинаковые имена разным файлам. Даже отдельный пользователь может присвоить одно и то же имя нескольким файлам.) Файлы с разными идентификаторами, но с одинаковым контентом считаются разными файлами. Должны ли одинаковые файлы разных владельцев храниться отдельно или целесообразно сэкономить память за счет хранения только одной копии файла? Чтобы обеспечить такую экономию, придется добавить уровень сложности для управления группами владельцев. Тогда любой из владельцев может видеть, что файл доступен другим владельцам, которые ему известны, но не пользователям, принадлежащим другим группам. Предполагается, что одинаковые файлы составляют небольшую долю всех файлов, так что такое решение может быть неоправданным. В исходном дизайне такие файлы будут храниться по отдельности, но следует помнить, что в проектировании систем не бывает непреложных истин (а следовательно, это искусство, а не наука). Можно обсудить с экспертом, что при достижении большого общего объема памяти, используемого CDN, экономия затрат за счет экономии памяти при дедупликации файлов может оправдать дополнительную сложность и затраты.

Размер файла может составлять несколько гигабайт или терабайт. Что, если выгрузка или загрузка завершится неудачей до своего окончания? Выгружать или загружать файл полностью заново неэффективно. Следует разработать процесс, сходный с механизмами создания контрольных точек или изоляции, для разбиения файла на блоки, чтобы клиенту приходилось повторять выгрузку или загрузку только незавершенных блоков. Такой процесс называется *отправкой по частям*.

Можно спроектировать протокол отправки по частям. В таком протоколе загрузка блока эквивалентна загрузке независимого файла. Для простоты примем, что блоки могут иметь фиксированный размер, например 128 Мбайт. Когда клиент начинает загрузку фрагмента, он отправляет исходное сообщение с обычными метаданными: идентификатором пользователя, именем файла и размером. Также в него может быть включен номер отправляемого блока. При отправке по частям хост хранения должен выделить подходящий диапазон адресов на диске для хранения файла и сохранения этой информации. Принимаемые блоки записываются по соответствующим адресам. Сервис метаданных отслеживает, загрузка каких блоков завершена. Когда клиент завершает загрузку последнего блока, сервис метаданных помечает файл как готовый к репликации и выгрузке. Если загрузка блока завершается неудачей, клиент может заново отправить только этот блок вместо всего файла.

Если клиент останавливает загрузку файла до того, как все блоки будут успешно приняты, эти блоки будут занимать место на хосте хранения. Можно реализовать

простое задание cron или пакетное задание ETl, периодически удаляющее эти блоки не полностью загруженных файлов. Другие возможные темы обсуждения:

- Возможность выбора размера блока клиентом.
- Репликация файла в процессе загрузки, чтобы файл быстрее стал доступным для выгрузки через CDN. Такое решение создает дополнительную сложность и вряд ли понадобится, но вы можете обсудить систему, в которой подобная высокая производительность обязательна.
- Клиент может запустить воспроизведение мультимедийного файла сразу же после выгрузки первого блока. Эта возможность кратко рассматривается в разделе 13.9.

ПРИМЕЧАНИЕ Отправка по частям с контрольными точками, которая обсуждается здесь, не имеет отношения к кодированию данных форм HTML по частям. Последнее предназначено для отправки данных форм, которые содержат файлы. За подробностями обращайтесь к таким источникам, как https://swagger.io/docs/speciication/describing-request-body/multipart-requests/ и https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST.

Другой вопрос — как организовать добавление, обновление (контента) и удаление файлов в такой распределенной системе. В разделе 4.3 обсуждается репликация операций обновления и удаления, возможные сложности и некоторые решения. Вы можете обсудить некоторые возможные решения, адаптированные из материала этого раздела:

- Решение с одним лидером, в котором конкретный датацентр назначается для выполнения этих операций и распространения изменений в другие датацентры. Такой подход может оказаться подходящим, особенно если не обязательно обеспечивать быструю доступность изменений во всех датацентрах.
- Решения с несколькими лидерами, включая кортежи. (Кортежи обсуждаются в книге Мартина Клеппмана «Designing Data-Intensive Systems».)
- Клиент устанавливает блокировку для этого файла в каждом датацентре, выполняет операцию в каждом датацентре, а затем снимает блокировку.

В каждом из этих решений фронтенд обновляет сервис метаданных информацией о доступности файла в датацентрах.

Не храните копии файлов во всех датацентрах

Некоторые файлы могут использоваться в основном в конкретных регионах (например, звуковые или текстовые файлы на определенных языках), так что хранить копии файлов во всех датацентрах может быть неэффективно. Чтобы определить, когда файл должен копироваться в определенный датацентр, можно использовать критерии репликации (например, количество запросов или

пользователей для этого файла в течение последнего месяца). Тем не менее это означает, что файл должен быть реплицирован в пределах датацентра для обеспечения отказоустойчивости.

Некоторые виды контента разделяются на несколько файлов из-за требований приложения к предоставлению определенных сочетаний файлов определенным пользователям. Например, видеофайл может предоставляться всем пользователям, и к нему будет прилагаться соответствующий аудиофайл на конкретном языке. Эта логика может обрабатываться на уровне приложения, а не в CDN.

Балансировка нагрузки пакетного задания etl

У нас есть периодическое (ежечасное или ежедневное) пакетное задание для распределения файлов между разными датацентрами и репликации файлов по соответствующему количеству хостов для удовлетворения спроса. Это пакетное задание получает журналы загрузки файлов за предыдущий период от сервиса ведения журналов, определяет значения счетчиков запросов файлов и использует эти показатели для регулировки численности хостов хранения для каждого файла. Затем оно создает карту файлов, которые должны быть добавлены или удалены из каждого узла, и использует эту карту для проведения соответствующих перестановок.

Для синхронизации в реальном времени можно разработать сервис метаданных, постоянно анализирующий местонахождения файлов и обращения к ним и перераспределяющий файлы.

Репликация между датацентрами — сложная тема, и скорее всего, вам не придется подробно обсуждать ее на собеседовании по проектированию систем, если только вы не претендуете на вакансию, требующую такой квалификации. В этом разделе рассматривается возможный дизайн обновления связей файлов в сервисе метаданных и файлов в сервисе хранения.

> **ПРИМЕЧАНИЕ** Узнать больше о настройке репликации между датацентрами в ZooKeeper можно из таких источников, как https://serverfault.com/questions/831790/how-to-manage-failover-in-zookeeper-across-datacenters-using-observers, https://zookeeper.apache.org/doc/r3.5.9/zookeeperObservers.html и https://stackoverflow.com/questions/41737770/how-to-deploy-zookeeper-across-multiple-data-centers-and-failover. Руководство по настройке репликации между датацентрами в Solr (поисковой платформе, использующей ZooKeeper для управления узлами) доступно по адресу https://solr.apache.org/guide/8_11/cross-data-center-replication-cdcr.html.

Обсудим подход к записи метаданных новых файлов в сервис метаданных, а также к соответствующему распределению файлов между датацентрами (внутрикластерная схема) или хостами (внекластерная схема) сервиса хранения. Создаваемый механизм должен отправлять запросы к сервису хранения для пересылки файлов между хостами разных датацентров. Чтобы предотвратить

возможное рассогласование между сервисом метаданных и сервисом хранения в случае неудачных запросов на запись, сервис метаданных должен обновлять только метаданные местонахождения своих файлов при получении успешного ответа от сервиса хранения, сообщающего, что файлы успешно записаны в новое расположение. Сервис хранения полагается на свой менеджер (внутрикластерный или внекластерный), обеспечивающий согласованность по своим узлам/хостам. Это гарантирует, что сервис метаданных не вернет расположение файлов до их успешной записи в новое место.

Более того, файлы должны удаляться с предыдущих узлов только после успешной записи в новое расположение. Таким образом, если запись файла в новое расположение завершается неудачей, файлы продолжают существовать в прежних расположениях, и сервис метаданных продолжит возвращать это расположение при получении запросов к этим файлам.

Можно использовать решение с сагой (см. раздел 5.6). На рис. 13.7 показано хореографическое решение, а на рис. 13.8 — оркестрованное решение, в котором сервис метаданных является оркестратором.

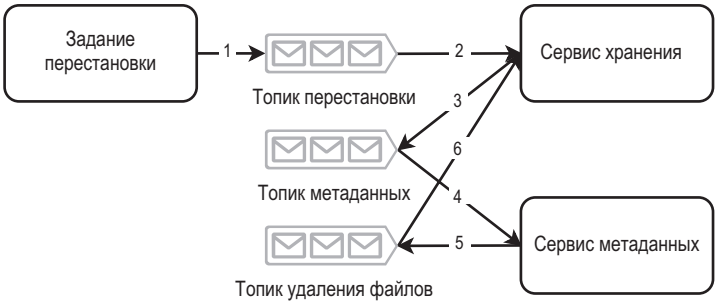

Рис. 13.7. Хореографическая сага для обновления сервиса метаданных и сервиса хранения

Последовательность шагов на рис. 13.7:

1. Задание перестановки отправляет событие в топик перестановки, что соответствует перемещению файла из одного расположения в другое. Это событие также может содержать такую информацию, как рекомендуемый коэффициент репликации файла, соответствующий количеству узлов-лидеров, на которых должен храниться этот файл.
2. Сервис хранения потребляет это событие и записывает файл в новое расположение.
3. Сервис хранения отправляет событие в топик метаданных, обращаясь с запросом к сервису метаданных для обновления метаданных расположения файла.

4. Сервис метаданных потребляет событие из топика метаданных и обновляет метаданные местонахождения файла.
5. Сервис метаданных отправляет событие в топик удаления файла, обращаясь с запросом к сервису хранения для удаления файлов из прежнего расположения.
6. Сервис хранения потребляет это событие и удаляет файлы из прежнего расположения.

Типы транзакций

Какие транзакции являются компенсируемыми, поворотными или транзакциями с возможностью повторения?

Все транзакции, предшествующие шагу 6, компенсируемые. Шаг 6 является поворотной транзакцией, потому что удаление файла необратимо. Это последний шаг, так что транзакции с возможностью повторения отсутствуют.

Тем не менее удаление файла можно реализовать как мягкое удаление (данные помечаются как удаленные, но реально не удаляются). Периодически можно выполнять жесткое удаление (удаление данных из физического хранилища без возможности использования или восстановления) из базы данных. В данном случае все транзакции являются компенсируемыми и поворотной транзакции нет.

На рис. 13.8 показаны следующие шаги:

1. Соответствует шагу 1 хореографического решения выше.
2. Сервис метаданных потребляет это событие.
3. Сервис метаданных отправляет событие в топик создания файлов, дает указание сервису хранения создать копии файла в новом расположении.
4. Сервис хранения потребляет это событие и записывает файлы в новое расположение.

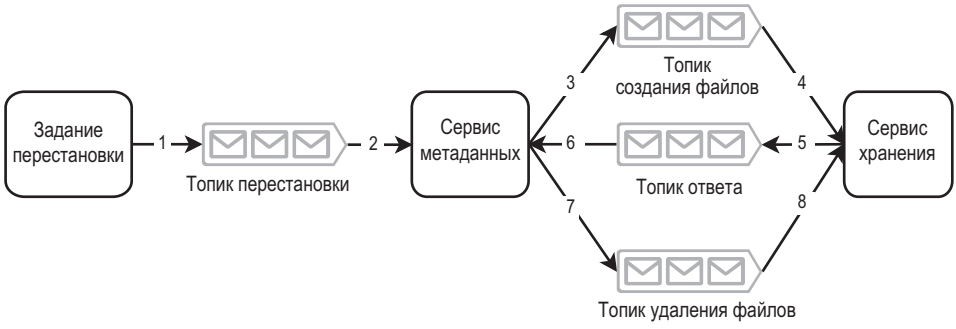

Рис. 13.8. Оркестрованная сага для обновления сервиса метаданных и сервиса хранения

5. Сервис хранения отправляет событие в топик ответа, сообщая сервису метаданных об успешном завершении записи файлов.
6. Сервис метаданных потребляет это событие.
7. Сервис метаданных отправляет событие в топик удаления файлов, давая указание сервису хранения удалить файл из прежнего расположения.
8. Сервис хранения потребляет это событие и удаляет файл из прежнего расположения.

13.7. ИНВАЛИДАЦИЯ КЭША

Так как CDN используется для статических файлов, инвалидация кэша не представляет серьезной проблемы. Можно добавить к файлам «цифровой отпечаток», как рассматривалось в разделе 4.11.1. Ранее мы обсуждали стратегии кэширования (рис. 4.8) и спроектировали систему для мониторинга кэша на устаревание файлов. Эта система должна предвидеть возможность высокого трафика.

13.8. ВЕДЕНИЕ ЖУРНАЛОВ, МОНИТОРИНГ И ОПОВЕЩЕНИЕ

В разделе 2.5 были рассмотрены ключевые концепции ведения журналов, мониторинга и уведомлений/оповещений, которые необходимо упомянуть на собеседовании. В дополнение к сказанному в разделе 2.5 отслеживать и отправлять оповещения необходимо в следующих случаях:

- Чтобы обеспечить возможность для отправителей отслеживать состояние своих файлов независимо от состояния отправки: в процессе, завершена успешно или завершилась неудачей.
- Для регистрации промахов при обращении к CDN с последующим мониторингом и выдачей некритичных оповещений для них.
- Для регистрации частоты запросов к файлам с помощью сервиса фронтенда. Это может делать общий сервис ведения журналов.
- В целях мониторинга аномальной или вредоносной активности.

13.9. ДРУГИЕ ВОЗМОЖНЫЕ ВОПРОСЫ ВЫГРУЗКИ ФАЙЛОВ МУЛЬТИМЕДИА

Возможно, вы захотите реализовать решение, в котором воспроизведение файлов мультимедиа начинается до завершения их выгрузки. Одно из возможных решений в этом случае — разбиение мультимедийных файлов на меньшие файлы. Такие файлы могут выгружаться последовательно и объединяться в файл

мультимедиа, являющийся частичной версией оригинала. Такая система требует, чтобы мультимедиа-плеер на стороне клиента выполнял сборку во время воспроизведения частичной версии. Подробности реализации этого решения выходят за рамки собеседования по проектированию систем. Оно подразумевает «сшивание» последовательности байтов из файлов.

Так как важно соблюдать последовательность действий, понадобятся метаданные, которые показывают, какие файлы должны выгружаться в первую очередь. Наша система разбивает файл на меньшие файлы и назначает каждому меньшему файлу номер в последовательности. Также можно сгенерировать файл метаданных, содержащий информацию о порядке файлов и их общем количестве. Как эффективно организовать выгрузку файлов в определенной последовательности? Можно обсудить и другие стратегии оптимизации потоковой передачи видео.

ИТОГИ

- CDN — масштабируемый стабильный сервис хранения файлов. Эта функциональность необходима практически любому веб-сервису, который обслуживает большую или географически распределенную базу пользователей.
- CDN — географически распределенный сервис хранения файлов, который позволяет каждому пользователю обратиться к файлу из датацентра, обслуживающего его запросы быстрее всех.
- Основные преимущества CDN — низкая задержка, масштабируемость, снижение удельной стоимости, повышение пропускной способности и доступности.
- К недостаткам CDN относятся дополнительная сложность, высокая удельная стоимость при низком трафике и скрытых затратах, дорогостоящая миграция, возможные сетевые ограничения, безопасность и соображения конфиденциальности, а также недостаточно широкие возможности настройки.
- Сервис хранения может быть отделен от сервиса метаданных, который отслеживает, на каких хостах сервиса хранения содержатся те или иные файлы. Реализация сервиса хранения строится на предоставлении хостов и поддержании работоспособности.
- Можно регистрировать обращения к файлам и использовать собранные данные для перераспределения или репликации файлов между датацентрами с целью оптимизации задержки и затрат памяти.
- CDN может использовать стороннюю аутентификацию и авторизацию на базе токенов с ротацией ключей для обеспечения безопасного, надежного и тонко настроенного управления доступом.
- В качестве высокоуровневой архитектуры CDN может использоваться типичная архитектура «шлюз API — метаданные — хранилище / база данных».

Каждый компонент настраивается и масштабируется под конкретные функциональные и нефункциональные требования.

- Управление сервисом распределенного хранения файлов может быть внутрикластерным или внекластерным. У каждого варианта есть свои плюсы и минусы.
- Файлы, к которым часто обращаются, могут кэшироваться на шлюзе API для ускорения чтения.
- Для управления ключами шифрования при хранении CDN может использовать сервис управления секретами.
- При загрузке больших файлов необходимо использовать механизм отправки по частям, при котором файл делится на блоки и каждый блок отправляется по отдельности.
- Чтобы обеспечивать низкую задержку выгрузки и управление затратами, периодическое пакетное задание может перераспределять файлы по датацентрам и выполнять их репликацию по соответствующему количеству хостов.

14 Проектирование приложения для обмена текстовыми сообщениями

В ЭТОЙ ГЛАВЕ

- ✓ Проектирование приложения для обмена короткими сообщениями между миллиардами клиентов
- ✓ Компромисс между задержкой и затратами
- ✓ Проектирование с расчетом на отказоустойчивость

Спроектируем приложение для обмена текстовыми сообщениями — систему для 100 тысяч пользователей, которые могут отправлять сообщения друг другу в течение нескольких секунд. Это не видео- или аудиочат. Пользователи отправляют сообщения с непредсказуемой частотой, так что система должна быть способна справиться с выбросами трафика. Это первый пример системы в книге, в котором рассматривается доставка по принципу «ровно один раз». Сообщения не должны теряться, но и не должны отправляться более одного раза.

14.1. ТРЕБОВАНИЯ

В ходе обсуждения можно определить следующие функциональные требования:

- Консистентность в реальном времени или в конечном счете? Рассмотрите оба варианта.

- Сколько пользователей могут присутствовать в чате? Чат может содержать от 2 до 1000 пользователей.
- Ограничен ли размер сообщения? Установим предел в 1000 символов UTF-8. При кодировании до 32 бит на символ одно сообщение может занимать до 4 Кбайт.
- Уведомления — платформенная подробность, которую можно не учитывать. В каждом из приложений для Android, iOS, Chrome и Windows может использоваться своя платформенно-зависимая библиотека уведомлений.
- Подтверждение доставки и прочтения.
- Хранение сообщений. Пользователи могут просматривать и проводить поиск в объеме до 10 Мбайт прошлых сообщений. Для 1 миллиарда пользователей это потребует до 10 Пбайт памяти.
- Тело сообщения является приватным. Можно обсудить с экспертом настройки доступа к просмотру какой-либо информации сообщения, включая информацию о том, что сообщение было отправлено от одного пользователя другому. Тем не менее события ошибок, например сбой отправки сообщения, должны инициировать ошибку, которая будет вам видна. Журнал ошибок и средства мониторинга должны сохранять конфиденциальность пользователя. Идеально подойдет решение со сквозным шифрованием.
- Рассматривать приток пользователей (то есть процесс регистрации новых пользователей в приложении) не нужно.
- Возможность создания нескольких чат-комнат (чат-румов)/каналов для одной группы пользователей не требуется.
- В некоторых чат-приложениях существуют шаблонные сообщения, которые пользователь может выбирать для быстрой отправки, например «Доброе утро!» или «Сейчас не могу говорить, отвечу позже». Это клиентская функциональность, которую мы рассматривать не будем.
- Некоторые приложения для обмена сообщениями дают возможность пользователю проверить, активно ли его соединение. Здесь эта возможность также не рассматривается.
- Мы разберем отправку только текста, без мультимедийных материалов: голосовых сообщений, фотографий, видео и т. д.

Нефункциональные требования:

- *Масштабируемость*: 100 тысяч одновременно обслуживаемых пользователей. Предположим, что каждый пользователь ежеминутно отправляет сообщение размером 4 Кбайт, что соответствует частоте записи 400 Мбайт/мин. Пользователь может иметь до 1000 соединений, и сообщение может отправляться до 1000 получателей, каждый из которых может иметь до 5 устройств.

- *Высокая доступность*: уровень «четыре девятки».
- *Высокая производительность*: значение параметра времени доставки 99-го процентиля — 10 секунд.
- *Безопасность и конфиденциальность*: необходима аутентификация пользователя. Сообщения должны быть приватными.
- *Консистентность*: строгое упорядочение сообщений не обязательно. Если несколько пользователей отправляют сообщения друг другу более или менее одновременно, эти сообщения могут отображаться в разном порядке для разных пользователей.

14.2. С ЧЕГО НАЧАТЬ

На первый взгляд может показаться, что задача напоминает сервис уведомлений/оповещений, рассмотренный в главе 9. Но если присмотреться внимательнее, мы увидим ряд различий, перечисленных в табл. 14.1. Наивная попытка использовать дизайн сервиса уведомлений/оповещений завершится неудачей, но можно воспользоваться им как отправной точкой. Сначала определим похожие требования и соответствующие компоненты дизайна, а затем воспользуемся отличиями для необходимого повышения или снижения сложности дизайна.

Таблица 14.1. Различия между приложением для обмена сообщениями и сервисом уведомлений/оповещений

Приложение для обмена сообщениями	Сервис уведомлений/оповещений
Все сообщения имеют одинаковые приоритеты и время доставки 99-го процентиля 10 секунд	События могут иметь разные уровни приоритета
Сообщения доставляются от одного клиента другим, все они находятся на одном канале на одном сервисе. Рассматривать другие каналы или сервисы не нужно	Несколько каналов (электронная почта, SMS, автоматизированные телефонные звонки, push-уведомления или уведомления в приложениях)
События инициируются только вручную	Событие может инициироваться вручную, на программном уровне или периодически
Шаблоны сообщений не поддерживаются (возможно, кроме предлагаемых сообщений)	Пользователи могут создавать шаблоны уведомлений и управлять ими
Из-за сквозного шифрования мы не можем видеть сообщения пользователя, что усложняет идентификацию и дедупликацию общих элементов в функции для сокращения потребления вычислительных ресурсов	Отсутствие сквозного шифрования. Пользователь имеет больше свободы для создания абстракций (например, сервисов шаблонов)

Приложение для обмена сообщениями	Сервис уведомлений/оповещений
Пользователи могут запрашивать старые сообщения	Большинство уведомлений достаточно отправить один раз
Подтверждение доставки и подтверждение прочтения являются частью приложения	Могут быть недоступны многие каналы уведомлений (электронная почта, текстовые сообщения, push-уведомления и т. д.), так что подтверждения доставки и прочтения могут быть невозможны

14.3. ИСХОДНАЯ ВЫСОКОУРОВНЕВАЯ АРХИТЕКТУРА

Сначала пользователь выбирает получателя своего сообщения (по имени) из списка получателей. Затем создает сообщение в приложении (мобильном, десктопном или браузерном), после чего нажимает кнопку отправки. Приложение сначала шифрует сообщение открытым ключом получателя, а затем отправляет запрос к сервису сообщений для доставки сообщений. Сервис сообщений отправляет сообщение получателю. Получатель отправляет подтверждение доставки и подтверждение прочтения отправителю. Такой дизайн имеет ряд следствий:

- Приложение должно хранить метаданные каждого получателя, включая имена и открытые ключи.
- Сервис сообщений должен поддерживать открытое соединение WebSocket к каждому получателю.
- Если получателей несколько, отправитель должен зашифровать сообщение открытым ключом каждого получателя.
- Сервис сообщений должен справляться с непредвиденными выбросами трафика от многих отправителей, неожиданно решивших отправить сообщения в течение короткого периода.

На рис. 14.1 показано создание разных сервисов для обслуживания разных функциональных требований и оптимизация их для нефункциональных требований:

- *Сервис отправителя*: получает сообщения от отправителей и немедленно доставляет их получателям. Также он записывает эти сообщения в сервис сообщений (см. ниже).
- *Сервис сообщений*: к нему отправители направляют запросы своих отправленных сообщений, а получатели — запросы как полученных, так и неполученных сообщений.
- *Сервис соединений*: используется для сохранения и получения активных и заблокированных соединений, добавления дополнительных пользователей в список контактов, блокировки определенных пользователей от отправки

сообщений. Сервис соединений также хранит метаданные соединений (например, имена, аватары и открытые ключи).

На рис. 14.1 показана высокоуровневая архитектура со связями между сервисами. Пользователи отправляют запросы к сервисам через шлюз API. Сервис-отправитель отправляет запросы к сервису сообщений о сохранении сообщений, включая сообщения, которые не удалось доставить получателю. Он также отправляет запросы к сервису соединений, чтобы проверить, не заблокировал ли получатель отправителя сообщения. Описанные действия более подробно рассматриваются в следующих разделах.

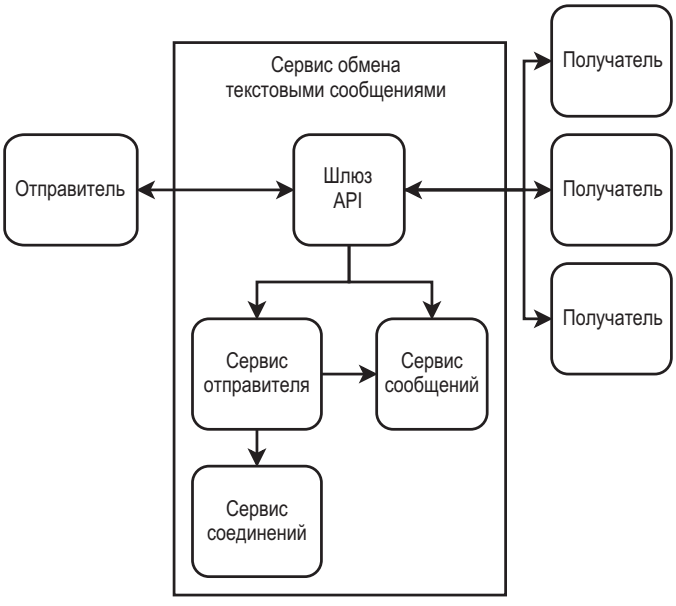

Рис. 14.1. Высокоуровневая архитектура со связями между сервисами. Получатель отправляет запросы отправки подтверждений доставки и прочтения, которые передаются отправителю. Любой пользователь (отправитель или получатель) может запросить у сервиса сообщений старые или недоставленные сообщения

14.4. СЕРВИС СОЕДИНЕНИЙ

Сервис соединений должен предоставлять следующие конечные точки:

- `GET /connection/user/{userId}}` — получение всех соединений пользователя (GET) и их метаданных, включая активные и заблокированные соединения, а также открытые ключи активных соединений. Также можно добавить дополнительные параметры пути или запроса для фильтрации групп соединений или других категорий.

- `POST /connection/user/{userId}: }/recipient/{recipientId}` — запрос нового соединения: пользователя `userId` с пользователем `recipientId` (запрос от `userId`).
- `PUT /connection/user/{userId}:/recipient/{ recipientId }/request/{accept}: accept` — логический признак принятия или отклонения запроса на соединение.
- `PUT /connection/user/{userId}:/recipient/{recipientId }/block/{block}: block` — логическая переменная для установления или снятия блокировки соединения.
- `DELETE /connection/user/{ userId}: /recipient/{recipientId}` — удаление соединения.

14.4.1. Создание соединений

Соединения пользователей (как активные, так и заблокированные) должны храниться на устройствах пользователей (то есть в их настольных или мобильных приложениях) либо в браузерных cookie или локальном хранилище, так что сервис соединений становится резервной копией для этих данных на случай смены устройства пользователем или синхронизации данных по нескольким устройствам пользователя. Мы не ожидаем интенсивного трафика записи или больших объемов данных, что позволяет реализовать его в виде простого сервиса бэкенда без сохранения состояния, который сохраняет данные в общем сервисе SQL.

14.4.2. Блокировка отправителя

Будем называть заблокированное соединение «заблокированным соединением отправителя», если пользователь заблокировал этого отправителя, и «соединением, заблокированным получателем», если пользователь заблокирован получателем. В этом разделе обсуждается подход к блокировке отправителей для максимизации производительности приложения и его офлайн-функциональности. На рис. 14.2 показано, что блокировка должна быть реализована на каждом уровне, то есть на стороне клиента (на устройствах и отправителя, и получателя) и на стороне сервера. В оставшейся части этого раздела рассматриваются некоторые важные особенности такого подхода.

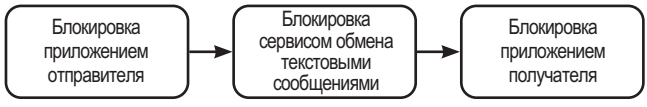

Рис. 14.2. Блокировка должна быть реализована на каждом уровне. Если отправитель заблокирован, его устройство должно блокировать попытки отправки сообщений. Если блокировка завершается неудачей и сообщение передается серверу, оно должно быть заблокировано сервером. Если и сервер не смог заблокировать сообщение и оно достигло устройства получателя, то это устройство должно заблокировать его

Сокращение трафика

Чтобы сократить трафик с сервером, соединения, заблокированные получателем, должны храниться на устройстве пользователя, чтобы устройство могло помешать пользователю взаимодействовать с этим получателем, а серверу не приходилось блокировать такие нежелательные взаимодействия. Нужно ли сообщать пользователю, что другой пользователь заблокировал его? Это решение из области UX-проектирования, которое остается на ваше усмотрение.

Разрешение немедленной блокировки/разблокировки

Когда пользователь отправляет запрос на блокировку отправителя, клиент также должен отправить соответствующий запрос PUT на блокировку отправителя. Однако на случай, если эта конкретная конечная точка временно недоступна, клиент может сохранить информацию о том, что он заблокировал отправителя, чтобы скрыть любые сообщения от заблокированного и не отображать уведомления о новых сообщениях. Для разблокирования отправителя выполняются те же действия. Эти запросы могут отправляться в очередь недоставленных сообщений на устройстве, а затем, когда конечная точка снова станет доступной, — на сервер. Это пример корректного сокращения функциональности. Ограниченная функциональность сохраняется даже в том случае, если часть системы становится недоступной.

Это означает, что другие устройства пользователя могут продолжать получать сообщения от предположительно заблокированного отправителя или блокировать сообщения от предположительно разблокированного отправителя. Сервис соединений следит, какие устройства синхронизировали свои соединения с нашим сервисом. Если синхронизированное устройство отправляет сообщение получателю, заблокировавшему отправителя, это указывает на возможную ошибку или вредоносную активность, и сервис соединений должен передать оповещение разработчикам. Эта тема более подробно обсуждается в разделе 14.6.

Взлом приложения

Помешать отправителю взломать приложение, если он захочет удалить данные о заблокировавших его получателях, не получится. Если шифровать данные заблокированных получателей на устройстве отправителя, единственным безопасным способом хранения ключей становится хранение на сервере; это означает, что устройство отправителя должно обращаться к серверу с запросом на просмотр заблокированных получателей, а это противоречит самой цели хранения данных на устройстве отправителя. Проблема безопасности — еще одна причина для реализации блокировки на каждом уровне. Подробное обсуждение безопасности и атак взлома выходит за рамки книги.

Возможная проблема с консистентностью

Пользователь может отправлять одинаковые запросы на блокировку или разблокировку с нескольких устройств. На первый взгляд кажется, что никаких проблем не будет, потому что запрос PUT идемпотентен. Однако это может привести к нарушению согласованности данных. Наш механизм корректного сокращения функциональности усложняет механизм блокировки. Если пользователь отправил запрос блокировки, а затем запрос разблокировки с одного устройства и одновременно отправил запрос блокировки с другого устройства, становится неясно, в каком же состоянии должен находиться отправитель в итоге — заблокированном или разблокированном (рис. 14.3)? Как уже не раз отмечалось, добавление метки времени устройства к запросу для определения порядка запросов проблемы не решает, так как часы устройств невозможно идеально синхронизировать.

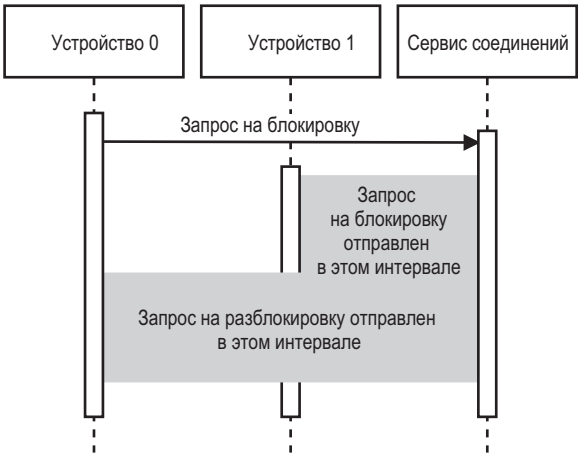

Рис. 14.3. Если сразу несколько устройств отправляют запросы сервису соединений, это может привести к нарушению консистентности. Если устройство 0 выдает запрос на блокировку, за которым следует запрос на разблокировку, а устройство 1 выдает запрос на блокировку после устройства 1, неясно, чего же хочет пользователь — заблокировать или разблокировать отправителя?

Разрешение на соединение только для одного устройства в определенный момент времени не решит проблему, потому что мы разрешаем ставить запросы в очередь на устройстве пользователя, если не удается отправить запрос к сервису. На рис. 14.4 пользователь создает соединение с одного устройства и отправляет запросы, которые ставятся в очередь, затем создает соединение с другого устройства и отправляет успешные запросы, после чего первое устройство успешно отправит запросы к серверу.

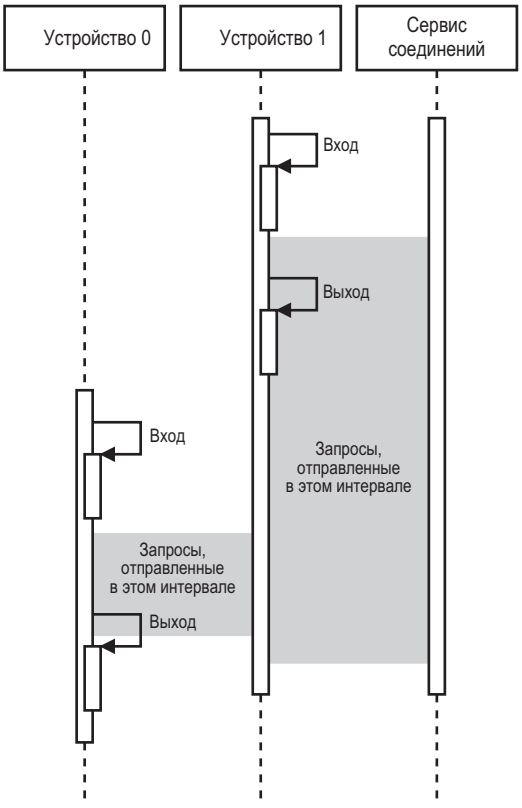

Рис. 14.4. Несогласованность возможна даже в том случае, если для отправки запроса устройство должно пройти процедуру входа, потому что запрос может быть поставлен в очередь на устройстве и будет отправлен сервису соединения после выхода пользователя из системы

Эта общая проблема согласованности проявляется, когда приложение предоставляет офлайн-функциональность, связанную с операциями записи.

Одно из возможных решений — предложить пользователю подтвердить итоговое состояние каждого устройства. (В частности, оно используется в приложении, написанном автором этой книги: https://play.google.com/store/apps/details?id=com.zhiyong.tingxie.) Последовательность действий может выглядеть так:

1. Пользователь выполняет операцию записи на одном устройстве, которое затем обновляет сервер.
2. Другое устройство синхронизируется с сервером и обнаруживает, что его состояние отлично от состояния на сервере.
3. Устройство открывает пользовательский интерфейс, который предлагает пользователю подтвердить итоговое состояние.

Другое возможное решение — установить ограничения на операции записи (и офлайн-функциональность), чтобы избежать возможной несогласованности.

В таком случае устройство, отправившее запрос на блокировку, не должно иметь разрешения снять ее, пока все остальные устройства не будут синхронизированы с сервером, — и наоборот для запросов на разблокировку.

Недостаток обоих решений заключается в том, что взаимодействие с пользователем усложняется. Появляется компромисс между удобством использования и согласованностью. Взаимодействие улучшится, если устройство сможет отправлять произвольные операции записи независимо от сетевого соединения, но в таком случае будет невозможно поддерживать консистентность данных.

Открытые ключи

Когда устройство устанавливает (или переустанавливает) ваше приложение, при первом запуске генерируется пара ключей — открытый и закрытый. Открытый ключ должен храниться в сервисе соединения. Сервис соединения должен немедленно обновлять соединения пользователя новым открытым ключом через соединения WebSocket.

Так как пользователь может иметь до 1000 соединений, каждое из которых поддерживает до 5 устройств, изменение ключей может потребовать до 5000 запросов, и некоторые запросы могут завершиться неудачей, потому что получатели могут оказаться недоступными. Скорее всего, изменения ключей будут относительно редкими, так что это не должно привести к непредсказуемым выбросам трафика, и сервису соединений не обязательно использовать брокеры сообщений или Kafka. Соединение, не получившее обновление, может получить его позже в запросе GET.

Если отправитель шифрует свое сообщение устаревшим открытым ключом, то после его расшифровки получатель получит бессмысленный набор символов. Чтобы устройство получателя не выводило такие ошибки, отправитель может хешировать сообщение криптографической хеш-функцией (например, SHA-2) и включить полученный хеш в сообщение. Устройство получателя может хешировать расшифрованное сообщение и выводить его получателю только в случае совпадения хешей. Сервис отправителя (более подробно рассматриваемый в следующем разделе) может предоставлять получателю специальную конечную точку, через которую получатель может потребовать у отправителя повторно отправить сообщение. Получатель может включить свой открытый ключ, чтобы отправитель не повторил ошибку и заменил устаревший открытый ключ новым.

Один из способов избежать таких ошибок — сделать так, чтобы изменение открытого ключа не вступало в силу немедленно. Запрос на изменение открытого ключа может включать период отсрочки (например, 7 дней), в течение которого действительными остаются оба ключа. Если получатель получает сообщение, зашифрованное со старым ключом, он отправляет сервису сообщений специальный запрос, а сервис сообщения дает указание отправителю обновить свой ключ.

14.5. СЕРВИС ОТПРАВИТЕЛЯ

Сервис отправителя оптимизирован для масштабируемости, доступности и производительности одной функции — отправки сообщений отправителей и их доставки получателям практически в реальном времени. Эта критическая функция должна быть максимально простой для оптимизации удобства ее отладки и обслуживания. При возникновении непредсказуемых выбросов трафика сообщения должны буферизоваться во временном хранилище, чтобы функция могла обработать эти сообщения и доставить их, когда у нее появятся достаточные ресурсы.

На рис. 14.5 изображена высокоуровневая архитектура сервиса отправителя. Он состоит из двух сервисов, между которыми находится топик Kafka. Назовем их сервисом новых сообщений и сервисом отправки сообщений. Такое решение напоминает сервис уведомлений из раздела 9.3. Однако здесь мы не используем сервис метаданных, потому что контент зашифрован и невозможно провести его парсинг, чтобы заменить общие компоненты идентификаторами.

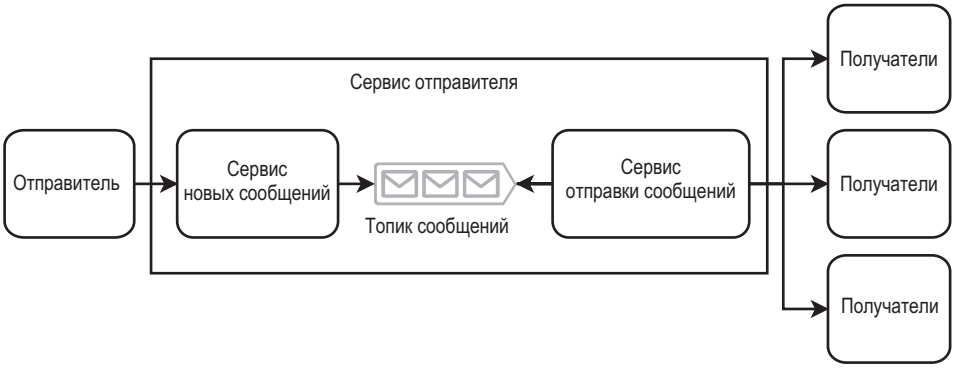

Рис. 14.5. Высокоуровневая архитектура сервиса отправителя. Отправитель отправляет сообщение сервису отправителя через шлюз API (не показан)

Сообщение содержит поля, в которых хранится идентификатор отправителя, список до 1000 идентификаторов получателей, строка тела и статус отправки из перечисления (возможные статусы — «сообщение отправлено», «Сообщение доставлено» и «Сообщение прочитано»).

14.5.1. Отправка сообщения

Отправка сообщения происходит по следующей схеме: на стороне клиента пользователь создает сообщение с идентификатором отправителя, идентификатором получателя и строкой тела. Подтверждение доставки и подтверждение прочтения инициализируются значением `false`. Клиент шифрует тело, а затем отправляет сообщение сервису отправителя.

Сервис новых сообщений получает запрос, отправляет его в топик новых сообщений Kafka, после чего возвращает отправителю код успеха 200. Запрос сообщения от одного отправителя может содержать до 5000 получателей, поэтому он должен обрабатываться асинхронно. Сервис новых сообщений также может выполнять простые проверки (например, на правильность форматирования запроса) и возвращать код ошибки 400 для недопустимых запросов (а также инициировать соответствующие оповещения для разработчиков).

На рис. 14.6 изображена высокоуровневая архитектура сервиса отправки сообщений. Генератор сообщений потребляет события из топика новых сообщений Kafka и генерирует отдельное сообщение для каждого получателя. Хост может породить новый поток или поддерживать пул потоков для генерации сообщения. Хост отправляет сообщение в топик Kafka, который мы назовем топиком получателя. Хост также может записать контрольную точку в распределенную базу данных в памяти (например, Redis). Если на хосте происходит сбой во время генерации сообщений, его заменитель может найти сохраненную контрольную точку, чтобы не генерировать дубликаты сообщений.

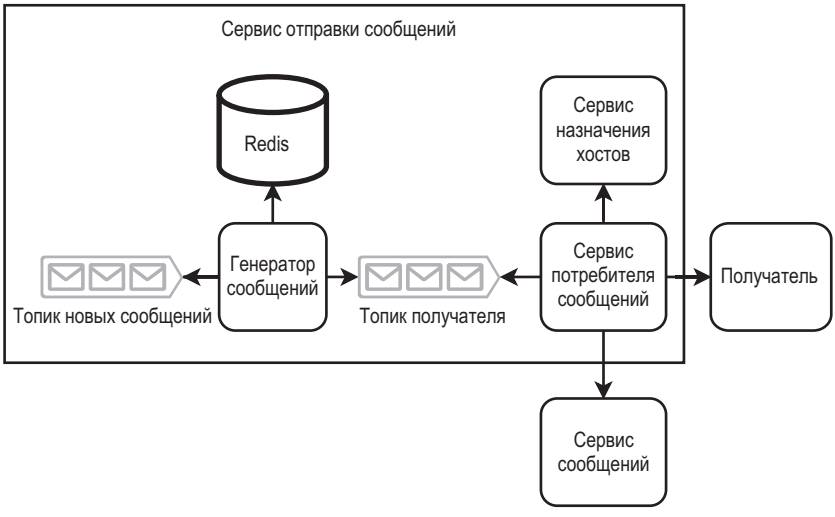

Рис. 14.6. Высокоуровневая архитектура сервиса отправки сообщений. Сервис потребителя сообщений может пользоваться другими сервисами для отправки сообщения получателю, а не отправлять его получателю напрямую, как показано на схеме

Сервис потребителя сообщений потребляет событие из топика получателя, а затем выполняет следующие действия:

1. Проверяет, был ли заблокирован отправитель. Сервис отправки сообщений должен хранить эти данные, а не отправлять запрос к сервису соединений

для каждого сообщения. Если у сообщения заблокирован отправитель, это означает, что в механизме блокировки на стороне клиента произошел сбой — возможно, из-за ошибок или вредоносной активности. В таком случае следует выдать оповещение для разработчиков.

2. Каждый хост сервиса отправки сообщений имеет соединения WebSocket с некоторым количеством получателей. Это количество можно настраивать для достижения оптимального баланса. Использование топика Kafka позволяет каждому хосту обслуживать большое количество получателей, так как он может потреблять из топика Kafka только тогда, когда будет готов доставить сообщение. Сервис может использовать сервис распределенной конфигурации (такой, как ZooKeeper) для назначения хостов устройствам. Этот сервис ZooKeeper может находиться за другим сервисом, который предоставляет конечные точки API для возвращения хоста, обслуживающего конкретного получателя. Назовем его сервисом назначения хостов.
   a) Хост сервиса отправки сообщений, который обрабатывает текущее сообщение, может запросить у сервиса назначения хостов нужный хост, а затем указать этому хосту доставить сообщение получателю. Подробнее см. в разделе 14.6.3.
   b) Параллельно сервис отправки сообщений также должен зарегистрировать сообщение в сервисе сообщений, который подробнее обсуждается в следующем разделе. Сервис отправки сообщений рассмотрен в разделе 14.6.
3. Сервис отправителя отправляет сообщение клиенту получателя. Если сообщение не может быть доставлено клиенту получателя (скорее всего, из-за того, что устройство получателя выключено или не имеет доступа в интернет), сообщение просто удаляется, поскольку оно уже зарегистрировано в сервисе сообщений и может быть загружено устройством позже.
4. Получатель проверяет, что сообщение не является дубликатом, а затем выводит его пользователю. При этом на устройстве пользователя может быть выдано уведомление от приложения получателя.
5. После того как пользователь прочитает сообщение, приложение может отправить отправителю подтверждениее прочтения, которое доставляется аналогично.

Шаги 1–4 показаны на диаграмме последовательности действий на рис. 14.7.

УПРАЖНЕНИЕ Как выполнить эти действия с хореографической или оркестрованной сагой? Нарисуйте соответствующие диаграммы хореографической или оркестрованной саги.

Как устройство может получить только неполученные сообщения? Один из вариантов, который приходит в голову, — регистрация сервисом сообщений информации о том, какие из устройств пользователя не получили сообщение.

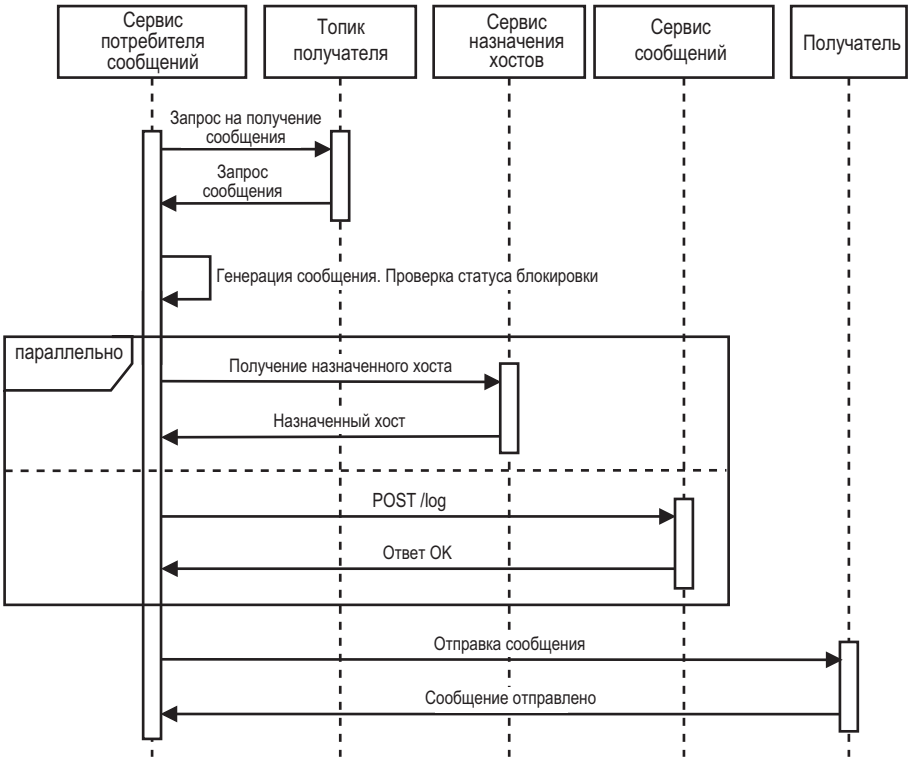

Рис. 14.7. Диаграмма последовательности действий: потребление сообщения из топика получателя Kafka и последующая отправка получателю

На основании этой информации для каждого устройства будут получены только непрочитанные сообщения. Такой подход предполагает, что сервису сообщений никогда не приходится доставлять одно и то же сообщение более одного раза на каждое устройство. Сообщения могут быть доставлены, а затем потеряны. Пользователь может удалить свои сообщения, а потом захотеть прочитать их снова. Приложение для обмена сообщениями может содержать ошибки, или же на устройстве могут возникнуть проблемы, из-за которых пользователь потеряет свои сообщения. Для такого сценария API сервиса сообщений может предоставить параметр пути или запроса, чтобы устройства могли запрашивать сообщения новее, чем последнее. При этом могут быть получены дубликаты сообщений, поэтому устройство должно проверить их наличие.

Как уже говорилось, сервис сообщений может иметь период удержания данных в несколько недель, по истечении которого сообщение удаляется.

Когда устройство получателя снова становится активным, оно может запросить у сервиса сообщений новые сообщения. Запрос будет направлен хосту, который

может запросить у сервиса метаданных новые сообщения, а затем вернуть их устройству получателя.

Сервис отправки сообщений также предоставляет конечную точку для обновления заблокированных/незаблокированных отправителей. Сервис соединений отправляет запросы к сервису отправки сообщений для обновления заблокированных/незаблокированных сообщений. Сервис соединений и сервис отправки сообщений разделены, чтобы иметь возможность независимого масштабирования; ожидается, что у второго трафик будет выше, чем у первого.

14.5.2. Другие темы для обсуждения

В ходе собеседования также можно рассмотреть следующие вопросы:

- Что произойдет, если пользователь отправит хосту бэкенда сообщение, но хост выйдет из строя до того, как вернет ответ, что сообщение получено?

Если хост бэкенда выйдет из строя, клиент получит ошибку 5xx. Для неудачных запросов можно реализовать такие стандартные средства, как повторные попытки с экспоненциальной задержкой, откат и очереди недоставленных сообщений. Клиент повторяет попытки, пока хост-производитель не поставит сообщение в очередь и вернет ответ 200 хосту бэкенда, а тот, в свою очередь, вернет ответ 200 отправителю.

На случай отказа хоста-потребителя можно реализовать автоматический или ручной процесс обработки отказов, чтобы другой хост-потребитель мог потребить сообщение из раздела Kafka, а затем обновить смещение этого раздела.

- Какой подход выбрать для определения порядка сообщений?

Можно воспользоваться согласованным хешированием, чтобы сообщения для конкретного получателя отправлялись в конкретный раздел Kafka. Это гарантирует, что сообщения для конкретного получателя будут потребляться и получаться по порядку.

Если метод согласованного хеширования приводит к тому, что некоторые разделы перегружаются сообщениями, можно увеличить количество разделов и изменить алгоритм согласованного хеширования, чтобы сообщения равномерно распределялись по большому количеству разделов. Другое возможное решение — использовать базу данных в памяти (например, Redis) для хранения маппинга (сопоставления) получателей с разделами и корректировать эти связи по мере надобности, чтобы предотвратить возможную перегрузку любого конкретного раздела.

Наконец, клиент также может проверять, что сообщения поступают по порядку. Если порядок нарушен, клиент может выдать некритичное оповещение о проблеме. Клиент также может удалить дубликаты сообщений.

- Что, если сообщения рассылаются по схеме «n:n / многие:многие» вместо 1:1?

Можно ограничить количество участников в чат-комнате.

Архитектура может масштабироваться. Масштаб можно как увеличивать, так и уменьшать с невысокими затратами. В архитектуре используются общие сервисы (такие, как шлюз API и общий сервис Kafka). Использование Kafka позволяет справляться с выбросами трафика без нарушения работоспособности.

Главным недостатком является задержка, особенно во время выбросов трафика. Использование пассивных механизмов (таких, как очереди) обеспечивает согласованность в конечном счете, но они не подходят для обмена сообщениями в реальном времени. Если в требованиях обозначено взаимодействие в реальном времени, воспользоваться очередями Kafka не получится и придется уменьшить соотношение между хостами и устройствами и поддерживать большой кластер хостов.

14.6. СЕРВИС СООБЩЕНИЙ

Сервис сообщений служит журналом регистрации сообщений. Пользователи могут обращаться к нему в следующих целях:

- Если пользователь только что вошел в приложение на новом устройстве или память приложения на устройстве была очищена, устройство должно загрузить свои последние сообщения (как отправленные, так и полученные).
- Доставка сообщения может оказаться невозможной. Среди причин — потеря питания устройством, отключение операционной системой или отсутствие сетевого соединения с устройством. Когда клиент будет включен, он может запросить у сервиса сообщений те, которые были отправлены ему, пока он был недоступен.

По соображениям конфиденциальности и безопасности система должна использовать сквозное шифрование, так что сообщения, проходящие через систему, зашифрованы. Другое преимущество сквозного шифрования заключается в том, что сообщения автоматически шифруются как во время передачи, так и при хранении.

Сквозное шифрование

Сквозное шифрование реализуется за три простых шага:

1. Получатель генерирует пару ключей (открытый и закрытый).
2. Отправитель шшфирует сообщение открытым ключом, после чего отправляет его получателю.
3. Получатель расшифровывает сообщение своим закрытым ключом.

После того как клиент успешно получит сообщения, сервис сообщений может удерживать данные в течение нескольких недель. По истечении этого срока сообщения удаляются для экономии места, а также повышения уровня конфиденциальности и безопасности. Удаление не позволяет взломщикам использовать возможные дефекты безопасности в сервисе для получения доступа к содержимому сообщений. Оно ограничивает объем данных, которые могут быть похищены и расшифрованы взломщиками, если им удастся получить закрытые ключи с устройств пользователей.

Но у пользователя может быть несколько устройств, на которых работает приложение обмена сообщениями. А если необходимо доставить сообщение на все устройства?

Один из вариантов — хранить сообщения в сервисе недоставленных сообщений и, возможно, запускать периодическое пакетное задание для удаления из очереди недоставленных сообщений данных возрастом больше заданного. Другой способ — разрешить пользователю вход в приложение только с одного телефона за раз и предоставить десктопное приложение, которое может отправлять и получать сообщения через телефон пользователя. Если пользователь входит с другого телефона, он не увидит свои старые сообщения с предыдущего телефона. Можно предоставить функциональность, которая позволяет пользователям архивировать свои данные в сервисе облачного хранения (например, Google Drive или Microsoft OneDrive), чтоы их можно было загрузить на другой телефон.

Наш сервис сообщений предполагает высокий трафик записи и низкий трафик чтения — идеальный сценарий использования для Cassandra. В архитектуру сервиса сообщений может быть включен сервис бэкенда без сохранения состояния и общий сервис Cassandra.

14.7. СЕРВИС ОТПРАВКИ СООБЩЕНИЙ

В разделе 14.5 обсуждался сервис отправителя, который содержит новый сервис сообщений для фильтрации недопустимых сообщений и последующей буферизации сообщений в топике Kafka. Основной объем обработки и доставки сообщений выполняется сервисом отправки сообщений, который более подробно рассматривается в этом разделе.

14.7.1. Введение

Чтобы сервис отправителя отправил сообщение получателю, последний сначала должен инициировать с ним сеанс связи, поскольку устройства получателя не являются серверами. В общем случае рассматривать пользовательские устройства как серверы недопустимо по следующим причинам:

- *Безопасность*: злоумышленники могут отправлять вредоносные программы на устройства (например, использовать их для DDoS-атак).
- *Повышенный сетевой трафик к устройствам*: устройства могут получать сетевой трафик от других сторон без предварительного создания сеанса. Из-за этого их владельцы могут нести дополнительные расходы за повышенный объем трафика.
- *Энергопотребление*: если каждое приложение будет требовать, чтобы устройство являлось сервером, повышенное энергопотребление значительно сократит время работы от батареи.

Можно воспользоваться P2P-протоколом, например BitTorrent, но для таких решений характерны компромиссы, рассмотренные выше. Мы не будем затрагивать эту тему.

Требование об инициировании соединений устройством означает, что сервис обмена сообщениями должен постоянно поддерживать большое количество соединений, по одному для каждого клиента. Для этого потребуется большой кластер хостов, что противоречит самой цели использования очереди сообщений. Технология WebSocket тоже не поможет, потому что открытые соединения WebSocket потребляют память хоста.

Кластеру-потребителю могут потребоваться тысячи хостов для обслуживания до 100 тысяч одновременных получателей/пользователей. Это означает, что каждый хост бэкенда должен поддерживать открытые соединения WebSocket с некоторым количеством пользователей, как показано на рис. 14.1. Наличие состояния неизбежно. Вам понадобится сервис распределенной координации, такой как ZooKeeper, для назначения хостов пользователям. Если хост выходит из строя, сервис ZooKeeper должен обнаружить это и предоставить хост-заменитель.

Рассмотрим процедуру отработки отказов при выходе из строя хоста сервиса отправки сообщений. Хост должен отправлять контрольные сигналы жизнеспособности на свои устройства. Если хост выходит из строя, его устройства запрашивают у сервиса отправки сообщений новые соединения WebSocket. Контейнерная система оркестрации (например, Kubernetes) должна предоставить новый хост, использовать ZooKeeper для определения его устройств и открыть соединения WebSocket с такими устройствами.

До утраты работоспособности старый хост мог успешно доставить сообщения некоторым получателям, но не всем. Как новому хосту избежать повторной доставки одного и того же сообщения и образования дубликатов?

Одно из решений — создание контрольной точки после каждого сообщения. Можно использовать базу данных в памяти (например, Redis) и секционировать кластер Redis для обеспечения сильной согласованности. Хост может выполнять запись в Redis каждый раз, когда сообщение успешно доставляется получателю.

Хост также читает данные из Redis, прежде чем доставлять сообщение, так что хост не будет доставлять дублирующиеся сообщения.

Другое возможное решение — просто заново отправить сообщения всем получателям в расчете на то, что устройства получателей удалят дубликаты сообщений.

В третьем варианте отправитель заново отправляет сообщение, если он не получил подтверждения в течение нескольких минут. Сообщение может быть обработано и доставлено другим хостом-потребителем. Если проблема не исчезнет, он отправляет оповещение в общий сервис мониторинга и оповещений, чтобы известить разработчиков об этой проблеме.

14.7.2. Высокоуровневая архитектура

На рис. 14.8 представлена высокоуровневая архитектура сервиса отправки сообщений. Основные компоненты сервиса:

1. Кластер обмена сообщениями — большой кластер хостов, каждый из которых назначается нескольким устройствам. Каждому отдельному устройству может быть назначен идентификатор.
2. Сервис назначения хостов — сервис бэкенда, использующий сервис ZooKeeper для поддержания маппинга (сопоставления) между идентификаторами устройств с хостами. Система управления кластерами (например, Kubernetes) также может использовать сервис ZooKeeper. В ходе отработки отказов Kubernetes обновляет сервис ZooKeeper, удаляет запись старого хоста и добавляет записи для новых предоставляемых хостов.
3. Сервис соединений, рассмотренный выше в этой главе.
4. Сервис сообщений, представленный на рис. 14.6. Каждое сообщение, полученное или отправленное устройству, также регистрируется в сервисе сообщений.

Каждый клиент соединяется с сервисом отправителей через WebSocket, так что хосты могут отправлять сообщения клиенту с задержкой почти реального времени. Это означает, что кластер обмена сообщениями должен содержать довольно большое количество хостов. Некоторым командам инженеров удалось создать миллионы одновременных соединений на одном хосте (https://migratorydata.com/2013/10/10/scaling-to-12-million-concurrent-connections-how-migratorydata-did-it/). Каждый хост также должен хранить открытые ключи своих соединений. Сервису обмена сообщениями понадобится конечная точка, через которую его соединения при необходимости могли бы отправлять своим хостам открытые ключи.

Тем не менее это не означает, что один хост может одновременно обрабатывать сообщения миллионов клиентов. Приходится идти на компромисс. Сообщения, которые могут быть доставлены за несколько секунд, должны быть небольшими, ограниченными несколькими сотнями символов. Можно создать отдельный

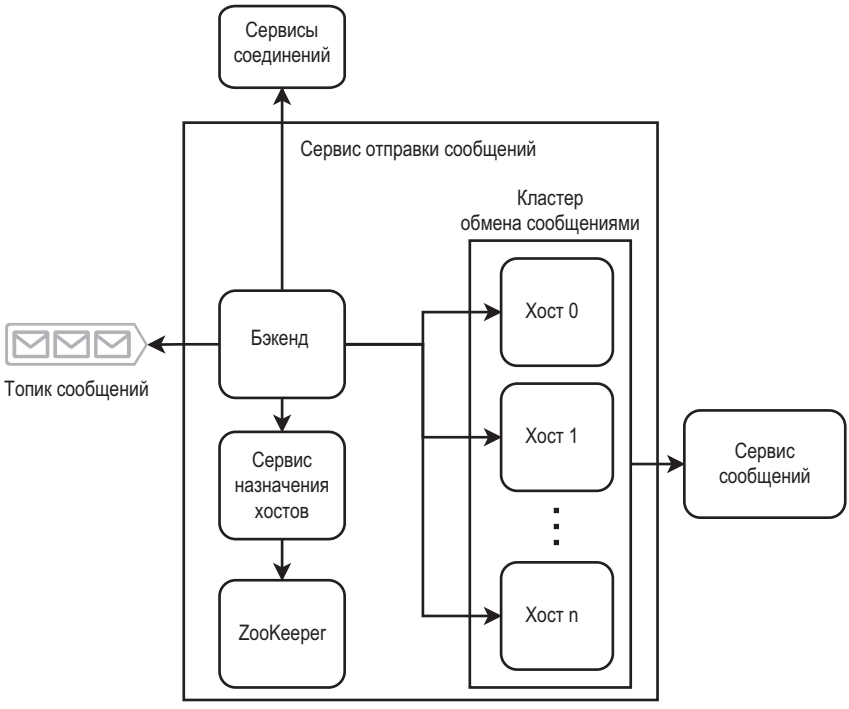

Рис. 14.8. Высокоуровневая архитектура сервиса отправки сообщений, связывающая клиентов с хостами. Резервное копирование сообщений на схеме не показано

сервис обмена сообщениями с собственным кластером хостов для работы с файлами (например, фото и видео) и масштабировать этот сервис независимо от сервиса обмена сообщениями, который работает с текстом. Во время выбросов трафика пользователи могут продолжать отправлять сообщения друг другу с задержкой в несколько секунд, но отправка файла может занять минуты.

Каждый хост может хранить сообщения до нескольких дней, периодически удаляя старые сообщения из памяти. На рис. 14.9 при получении сообщения хост сохраняет его в памяти, одновременно порождая поток для отправки сообщения в очередь Kafka. Кластер-потребитель потребляет сообщение из очереди и записывает его в общий сервис Redis. (Запись в Redis выполняется быстро, но также можно буферизовать запись средствами Kafka для повышения отказоустойчивости.) Когда клиент запрашивает старые сообщения, запрос проходит через бэкенд к хосту, а хост читает старые сообщения из общего сервиса Redis. При таком подходе чтение приоритетнее записи, так что запросы чтения будут иметь низкую задержку. Более того, так как трафик записи будет намного выше трафика чтения, использование очереди Kafka гарантирует, что выбросы трафика не перегрузят сервис Redis.

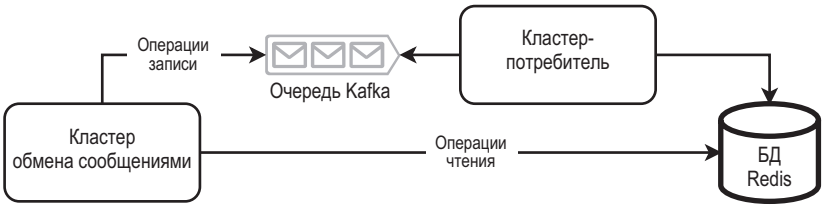

Рис. 14.9. Взаимодействие между кластером обмена сообщениями и базой данных Redis. Можно использовать очередь Kafka для буферизации чтения, чтобы повысить уровень отказоустойчивости

Сервис назначения хостов может включать маппинг идентификаторов клиентов / чат-комнат с хостами, которые хранятся в Redis. Для назначения идентификаторов хостам можно использовать согласованное хеширование, циклический перебор или взвешенный циклический перебор, но это может быстро привести к проблеме перегрева сегментов (некоторые хосты получают непропорциональную нагрузку). Сервис метаданных содержит информацию о трафике каждого хоста, а сервис назначения хостов на основании этой информации принимает решения, какой хост назначить клиенту или чат-комнате для предотвращения проблемы перегрева. Хосты можно перебалансировать так, чтобы каждый хост обслуживал одинаковые доли клиентов с высоким трафиком и клиентов с низким трафиком.

Сервис метаданных также хранит информацию об устройствах каждого пользователя.

Хост регистрирует свою активность запросов (то есть активность обработки при передаче сообщений) в сервисе ведения журналов, который хранит информацию в HDFS. Запуск периодического пакетного задания перераспределяет нагрузку между хостами, переназначая клиенты и хосты и обновляя сервис метаданных. Для лучшей балансировки нагрузки можно применить и более сложные статистические методы, например машинное обучение.

14.7.3 Последовательность действий при отправке сообщения

Рассмотрим шаг 2а из раздела 14.5.1 более подробно. Когда сервис бэкенда отправляет сообщение другому индивидуальному устройству или в чат-комнату, для текста и файлов из этого сообщения в отдельности выполняются следующие действия:

1. Хост бэкенда отправляет запрос к сервису назначения хостов, который при помощи ZooKeeper определяет, какие хосты обслуживают получателя — отдельный клиент или чат-комнату. Если хост еще не назначен, ZooKeeper может его назначить.

2. Хост бэкенда отправляет сообщение этим хостам, которые мы будем называть «хостами-получателями».

14.7.4. Некоторые вопросы

Можно ожидать, что эксперт задаст вопросы о состоянии. Описанная архитектура противоречит принципам облачных платформ, согласно которым в приоритете находится консистентность в конечном счете. Можно обсудить, что она не подходит для приложений обмена сообщениями, особенно для групповых чатов. Для облачных платформ характерны определенные компромиссы (например, более высокая задержка записи и консистентность в конечном счете ради низкой задержки чтения, более высокой доступности и т. д.), которые не совсем подходят нашим требованиям низкой задержки чтения и сильной консистентности. Какие еще вопросы можно обсудить на собеседовании:

- *Что произойдет, если сервис выйдет из строя, прежде чем доставит сообщение получателю или уведомление об отправке сообщения отправителю?* Мы уже обсуждали, что делать, если какое-либо из устройств получателя недоступно. Как гарантировать доставку отправителю уведомлений об отправке? Одно из решений — хранение недавних событий отправки на хостах клиента и получателя. Для этого хорошо подойдет технология Cassandra благодаря ее быстрой записи. Если отправитель не получил сообщение через какое-то время, он обращается с запросом к сервису обмена сообщениями, чтобы определить, было ли сообщение отправлено. Хост клиента или хост получателя возвращает ответ об успехе отправителю. Также можно рассматривать уведомление об отправке как отдельное сообщение. Хост получателя отправляет его на устройство отправителя.
- *Какой способ использовать для упорядочения сообщений*? Каждое сообщение должно иметь метку времени, полученную от клиента отправителя. Может оказаться, что более поздние сообщения будут успешно обработаны и доставлены до более ранних сообщений. Если устройство получателя выводит сообщения по порядку, а пользователь просматривает свое устройство, более ранние сообщения неожиданно могут появиться до более поздних, что создаст путаницу у пользователя. Одно из возможных решений — игнорировать более раннее сообщение, если более позднее уже было доставлено на устройство получателя. Когда клиент получателя получает сообщение, он проверяет наличие сообщений с более поздними метками времени, и если находит их, возвращает ошибку 422 с подходящим описанием. Ошибка может распространиться на устройство отправителя. Пользователь, отправивший сообщение, может решить отправить сообщение повторно, зная, что оно появится после успешно доставленного более позднего сообщения.
- *Что, если сообщения рассылаются по схеме «n:n / многие:многие», а не 1:1?* Необходимо ограничить количество людей в чат-комнате.

14.7.5. Повышение доступности

В высокоуровневой архитектуре на рис. 14.8 каждый клиент связывается с одним хостом. Даже если существует сервис мониторинга, который получает контрольные сигналы жизнеспособности от хостов, восстановление после отказа хоста займет не менее десятков секунд. Сервису назначения хостов необходимо выполнить сложный алгоритм для перераспределения клиентов между хостами.

Доступность можно улучшить созданием пула резервных хостов, которые обычно не обслуживают клиенты, а только отправляют контрольные сигналы жизнеспособности. При отказе хоста сервис назначения хостов немедленно переводит все свои клиенты на резервный хост. Такое решение сокращает время простоя до секунд; можете обсудить с экспертом, приемлемо ли оно.

Дизайн, минимизирующий время простоев, основан на создании мини-кластеров. Каждому хосту (назовем его первичным хостом) назначается один или два вторичных хоста. Первичный хост будет постоянно перенаправлять все свои запросы своим вторичным хостам; это гарантирует, что вторичные хосты будут синхронизироваться по первичному хосту и всегда будут готовы принять роль первичного хоста. Когда на первичном хосте происходит сбой, отработка отказа с переходом на вторичный хост осуществляется немедленно. Для определения этой инфраструктуры можно использовать Terraform. Определите кластер Kubernetes из трех модулей, по одному узлу на каждый модуль. Возможно, в целом это решение окажется слишком затратным и сложным.

14.8. ПОИСК

Каждый пользователь может выполнять поиск только в своих сообщениях. Можно реализовать прямой поиск в текстовых сообщениях и отказаться от построения обратного индекса в каждом клиенте, избегая затрат на проектирование, реализацию и обслуживание обратного индекса. Скорее всего, размер хранилища сообщений среднего клиента будет намного меньше 1 Гбайт (если не считать файлы мультимедиа). Загрузить сообщения в память и провести в них поиск относительно несложно. Можно выполнять поиск по именам файлов мультимедиа, но не по содержимому самих файлов. Поиск в байтовых последовательностях выходит за рамки книги.

14.9. ВЕДЕНИЕ ЖУРНАЛОВ, МОНИТОРИНГ И ОПОВЕЩЕНИЯ

В разделе 2.5 обсуждались ключевые концепции ведения журналов, мониторинга и оповещений, которые необходимо упомянуть на собеседовании. В дополнение к сказанному в разделе 2.5 отслеживать и отправлять оповещения необходимо в следующих случаях:

- Регистрация запросов между сервисами (например, от шлюза API к сервису бэкенда).
- Регистрация событий отправки сообщения. Чтобы сохранить конфиденциальность пользователя, можно регистрировать не всю доступную информацию.
- Для обеспечения конфиденциальности пользователей никогда не регистрируйте содержимое сообщения, включая все его поля (отправитель, получатель, тело, подтверждение доставки и подтверждение прочтения).
- Регистрация информации о том, было ли сообщение отправлено внутри датацентра или между датацентрами.
- Регистрация событий ошибок (например, ошибок при отправке сообщений, событий подтверждения доставки и событий подтверждения чтения).

Кроме сказанного в разделе 2.5, следует проводить мониторинг и отправлять оповещения для следующих ситуаций:

- Как обычно, следует отслеживать ошибки и тайм-ауты разных сервисов в целях принятия решений по масштабированию. Потребление памяти отслеживается в сервисе недоставленных сообщений.
- Комбинация отсутствия ошибок в сервисе бэкенда и стабильно малых затрат памяти в сервисе недоставленных сообщений указывает на то, что, возможно, стоит задуматься над снижением размера кластера сервиса отправителей.
- Также следует отслеживать возможные аномальные ситуации и признаки мошенничества, например отправку клиентом сообщений с аномально высокой частотой. Отправка программными средствами запрещена. Рассмотрите возможность установки ограничителя частоты запросов перед шлюзом API или сервисом бэкенда. Полностью блокируйте такие клиенты от отправки или получения сообщений на то время, пока вы разбираетесь с проблемой.

14.10. ДРУГИЕ ВОЗМОЖНЫЕ ТЕМЫ

Еще несколько возможных тем обсуждения, касающихся этой системы:

- Чтобы один пользователь мог отправлять сообщения другому, первый должен сначала запросить разрешение у второго. Последний может разрешить или заблокировать отправку. Позже он может изменить решение и дать разрешение после блокировки.
- Пользователь может заблокировать другого пользователя в любой момент, и тогда последний не сможет выбрать его для отправки сообщений. Такие пользователи не могут находиться в одной чат-комнате. Заблокировав пользователя, вы удаляете себя из любых чат-комнат, в которых присутствует этот пользователь.

- Разрешать ли вход с другого устройства? Следует разрешить вход только с одного устройства за раз.
- Система не гарантирует, что сообщения будут приняты в том порядке, в каком они были отправлены. Более того, если в чат-комнате несколько участников, которые отправляют сообщения почти одновременно, другие участники могут получить сообщения не по порядку. Сообщения могут поступать в разном порядке для разных участников. Как спроектировать систему, которая гарантирует, что сообщения отображаются по порядку? Из каких предположений при этом исходить? Если участник A отправляет сообщение, пока его устройство не подключено к интернету, а другие участники, имеющие подключение, отправляют сообщения вскоре после этого, в каком порядке должны выводиться сообщения на других устройствах и в каком — на устройстве участника A?
- Как расширить систему для поддержки файловых вложений или аудио/видеочатов? Вы можете кратко обсудить новые компоненты и сервисы.
- Мы не обсуждали удаление сообщений. В типичном приложении для обмена сообщениями пользователи имеют возможность удалять сообщения, после чего уже не смогут получить их снова. Необходимо разрешить пользователям удалять сообщения даже в том случае, если их устройство находится офлайн и удаление должно синхронизироваться с сервером. Можно более подробно обсудить такой механизм синхронизации.
- Можно подробнее обсудить механизм блокировки или разблокировки пользователей.
- Какие риски безопасности и конфиденциальности существуют в текущем дизайне? Как от них избавиться?
- Как система поддерживает синхронизацию по нескольким устройствам пользователя?
- Какие ситуации гонки могут возникнуть, когда пользователь добавляет в чат других пользователей или удаляет их из чата? Что, если произойдет скрытая ошибка? Как сервис может обнаруживать и устранять несогласованность?
- Мы не обсуждали системы обмена сообщениями, основанные на протоколах P2P (Peer-to-Peer), такие как Skype или BitTorrent. Так как клиент использует динамический, а не статический IP-адрес (статический IP-адрес — платная услуга, предоставляемая интернет-провайдером клиента), клиент может запустить демон, обновляющий сервис при изменении IP-адреса. Какие это может создать затруднения?
- Чтобы сократить вычислительные ресурсы и затраты, отправитель может сжать сообщение, прежде чем шифровать и отправлять его. Получатели могут восстановить сжатое сообщение после того, как оно будет получено и расшифровано.

- Обсудите дизайн системы, касающийся введения новых пользователей. Как новый пользователь может присоединиться к приложению для обмена сообщениями? Как он может добавлять или приглашать свои контакты? Пользователь может добавлять контакты вручную или с использованием Bluetooth или QR-кодов. Или же мобильное приложение может обратиться к списку контактов на телефоне, что потребует соответствующих разрешений Android или iOS. Пользователи могут приглашать других пользователей, отправляя им URL для загрузки или регистрации в приложении.
- Наша архитектура централизованная. Каждое сообщение должно пройти через бэкенд. Также можно обсудить децентрализованную структуру, например архитектуру P2P, в которой каждое устройство является сервером и может получать запросы от других устройств.

ИТОГИ

- Главная тема в обсуждении дизайна простой системы обмена текстовыми сообщениями — как организовать маршрутизацию многих сообщений между многими клиентами.
- Система чата напоминает сервис уведомлений/оповещений. Оба сервиса отправляют сообщения большому количеству получателей.
- Масштабируемый и эффективный с точки зрения затрат метод обработки выбросов трафика — использование очередей сообщений. С другой стороны, выбросы трафика повышают задержку.
- Задержку можно уменьшить за счет сокращения числа пользователей на хосте, однако это приводит к увеличению затрат.
- Каждое решение должно обрабатывать отказы хостов и переназначать пользователей хоста на другие хосты.
- Устройство получателя может быть недоступно, поэтому следует предоставить конечную точку GET для получения сообщений.
- Необходимо регистрировать запросы между сервисами и подробности событий отправки сообщений и событий ошибок.
- Отслеживание метрик использования позволит отрегулировать размер кластера и вести мониторинг попыток мошенничества.

15 Проектирование Airbnb

В ЭТОЙ ГЛАВЕ

- ✓ Проектирование системы бронирования
- ✓ Проектирование систем управления заявками и бронированием, предназначенных для групп эксплуатации
- ✓ Масштабирование сложной системы

Вам ставят задачу спроектировать сервис краткосрочной аренды жилья. Эта задача относится и к области программирования, и к области проектирования систем. Вопросы программирования будут обсуждаться при написании кода для решения, состоящего из набора классов, в стиле объектно-ориентированного программирования (ООП). Будем считать, что данная задача применима к системам бронирования вообще, в том числе:

- билетов в кино;
- авиабилетов;
- парковочных мест;
- такси или совместных поездок, хотя в этом случае нефункциональные требования и дизайн системы отличаются.

15.1. ТРЕБОВАНИЯ

Прежде чем рассматривать требования, можно обсудить, какую же систему нужно спроектировать. Приложение Airbnb выполняет следующие функции:

1. Функцию бронирования, поэтому в нем поддерживается тип пользователей, которые совершают бронь элемента из конечного набора. Airbnb называет таких пользователей «гостями». Также существует тип пользователей, которые создают такие наборы элементов. Airbnb называет их «хозяевами».
2. Функцию маркетплейса, сводящего продавцов товаров и услуг с покупателями. Airbnb сводит гостей с хозяевами.
3. Функцию обработки платежей и сбора комиссии. Это означает, что существуют внутренние пользователи, которые обеспечивают клиентскую поддержку и эксплуатацию для урегулирования споров, мониторинга и реакции на мошеннические действия. Этим Airbnb отличается от более простых приложений, таких как Craigslist. Большинство сотрудников таких компаний, как Airbnb, занято в сферах поддержки пользователей и эксплуатации.

На этой стадии можно уточнить у эксперта, ограничивается ли область применения хозяевами и гостями или же включает и другие типы пользователей. В этой главе будут обсуждаться хозяева, гости, сотрудники группы эксплуатации (далее операторы) и аналитики.

Ниже перечислены сценарии использования для хозяев. Список может быть очень длинным, поэтому мы ограничимся следующим:

- Регистрация гостей и обновление объявлений (добавление, редактирование, удаление). Обновления могут включать такие небольшие задачи, как изменение фотографий, но возможна и более сложная бизнес-логика. Например, объявление может иметь минимальную и/или максимальную продолжительность бронирования, а цены могут изменяться по дням недели или другим критериям. Приложение может выводить рекомендации по ценам. Объявления могут подчиняться местному законодательству. Например, законы краткосрочной аренды жилья в Сан-Франциско ограничивают продолжительность аренды при отсутствии хозяина в жилье 90 днями в год. Некоторые изменения в объявлениях также могут требовать подтверждения от операторов перед публикацией.
- Бронирование (например, принятие или отклонение заявок на бронирование):
  - Хозяин может иметь возможность просматривать рейтинг гостя и отзывы других хозяев, прежде чем принимать или отклонять заявку на бронирование от этого гостя.

  - Airbnb может предоставлять дополнительные возможности, например автоматическое принятие заявок, соответствующих критериям, заданным хозяином (например, от гостя с высоким рейтингом).
  - Отмена заявки после ее принятия. Отмена может приводить к денежным штрафам или временному запрету на публикацию предложений аренды. Конкретные правила могут быть довольно сложными.
- Коммуникация с гостями (например, через систему обмена сообщениями в приложении).
- Публикация оценок и отзывов о гостях, просмотр опубликованных оценок и отзывов.
- Получение оплаты от гостей (за вычетом комиссии Airbnb).
- Получение документов для оформления налоговых деклараций.
- Аналитика: просмотр изменений доходов, оценок и отзывов по времени.
- Коммуникация с операторами, включая запросы на разрешение споров (например, требований о возмещении ущерба гостями) или сообщения о мошенничестве.

Основные сценарии использования для гостей:

- Поиск и просмотр объявлений.
- Отправка заявок на бронирование, оплата и проверка статуса заявок на бронирование.
- Коммуникация с хозяевами.
- Отправка оценок и отзывов по приложениям, просмотр оценок и отзывов хозяев.
- Коммуникация с операторами (то же, что для хозяев).

Основные сценарии использования для операторов:

- Проверка запросов на размещение объявлений и удаление недопустимых объявлений.
- Коммуникация с пользователями: разрешение споров, предложение других вариантов размещения, возврат средств.

Мы не будем подробно обсуждать платежи, потому что это очень сложная тема. Платежное решение должно работать с разными валютами и законодательством (включая налоговое), различающимся в зависимости от страны, штата, города и т. д., а также в зависимости от продуктов и сервисов. Можно установить разные размеры комиссий в зависимости от типа платежа (например, максимальную сумму платежа по чекам или скидку для платежей по подарочным картам в целях

стимулирования их продаж). Механизмы и правовые нормы возмещения средств зависят от типа платежа, продукта, страны, клиента и множества других факторов. Существуют сотни и даже тысячи способов платежей, в числе которых:

- наличные;
- разные сервисы дебетовых и кредитных карт: MasterCard, Visa и многие другие. Каждый сервис имеет собственный API;
- системы онлайн-платежей, такие как PayPal или Alipay;
- чеки;
- товарный кредит;
- платежные и подарочные карты, которые могут быть привязаны к конкретным комбинациям компаний и стран;
- криптовалюты.

После примерно 5–10 минут краткого обсуждения требований и создания набросков дизайна уточните следующие функциональные требования:

- Хозяин публикует объявление о сдаче жилья в аренду. Предполагается, что объект размещения предназначен для одного гостя. Основные параметры объекта — город и цена. Хозяин может добавить до 10 фотографий и видеоролик объемом до 25 Мбайт.
- Гость может фильтровать объекты по городам, дате заезда и выезда.
- Гость может забронировать объект с датами заезда и выезда. Подтверждение хозяина для бронирования не требуется.
- Хозяин или гость может отменить бронь в любое время до ее начала.
- Хозяин или гость может просматривать свой список бронирования.
- Гость может забронировать только один объект в указанные даты.
- Объекты не могут быть забронированы двумя гостями одновременно.
- Для простоты в отличие от реального Airbnb не поддерживаются следующие функции:
  - возможность для хозяев вручную принимать или отклонять заявки на бронирование;
  - отмена бронирования (гостем или хозяином) после его совершения;
  - уведомления (такие, как push-уведомления или электронные письма) для гостей или хозяев можно обсудить кратко, но не вдаваться в подробности;
  - обмен сообщениями между пользователями, например между гостями и хозяевами или между операторами и гостями/хозяевами.

Следующие темы выходят за рамки нашей задачи. Эти возможные функциональные требования полезно упомянуть, чтобы продемонстрировать свой уровень критического мышления и внимания к деталям.

- Другие подробности объекта размещения:
  - Точный адрес. Обязательна только строка города. Другие подробности (штат, страна) игнорируются.
  - Предполагается, что для каждого объекта разрешен только один гость.
  - Тип объекта — квартира целиком, отдельная комната или спальное место в комнате.
  - Бытовые подробности — отдельный/общий санузел, оснащение кухни и т. д.
  - Возможность заселения с детьми.
  - Возможность заселения с домашними животными.
- Аналитика.
- Airbnb предоставляет хозяевам рекомендации по ценообразованию. В предложении можно установить минимальную и максимальную цену за ночь, и Airbnb изменит цену в указанном диапазоне.
- Дополнительные платежи и условия (например, плата за уборку и прочие платежи), особая цена в пиковые даты (например, выходные и праздники) или налоги.
- Платежи или возврат средств, включая штрафы за отмену брони.
- Поддержка клиентов, включая разрешение споров. Хороший уточняющий вопрос — нужно ли обсуждать, как оператор обрабатывает запросы на размещение объявлений. Также можно спросить, ограничивается ли поддержка клиентов, выходящая за рамки обсуждения, процессом бронирования или включает процесс размещения объявлений. Можно уточнить, что термин «клиент» в данном случае относится как к хозяевам, так и к гостям. Мы предположим, что эксперт попросит кратко обрисовать механизм проверки объявления операторами.
- Страховка.
- Чат или другие средства коммуникации любых сторон, например хозяина и гостя. Эта тема выходит за рамки нашего обсуждения, поскольку относится к сервисам обмена сообщениям или сервисам уведомлений (которые мы рассмотрели в других главах), а не к сервису бронирования.
- Регистрация и вход в систему.
- Компенсация хозяевам и гостям за неработоспособность сервиса.
- Отзывы пользователей: например отзыв гостя о проживании или отзыв хозяина о поведении гостя.

Конечные точки API для публикации предложений и бронирования могут выглядеть так:

- `findRooms(cityId, checkInDate, checkOutDate)`;
- `bookRoom(userId, roomId, checkInDate, checkOutDate)`;
- `cancelBooking(bookingId)`;
- `viewBookings(hostId)`;
- `viewBookings(guestId)`.

Нефункциональные требования могут выглядеть так:

- Масштабируемость до 1 миллиарда объектов размещения или 100 миллионов ежедневных бронирований. Истекшие бронирования могут удаляться. Программная генерация пользовательских данных запрещена.
- Сильная консистентность для бронирования, а точнее, доступности объявлений, чтобы избежать двойного бронирования или бронирования на недоступные даты. Для другой информации (например, описаний или фотографий) может быть приемлема консистентность в конечном счете.
- Высокая доступность, поскольку потеря бронирования приводит к упущенной выгоде. Но как будет объяснено в разделе 15.2.5, невозможно полностью предотвратить потери бронирования, если требуется избежать двойных бронирований.
- Высокая производительность не обязательна. Задержка в несколько секунд для 99-го процентиля приемлема.
- Стандартные требования к безопасности и конфиденциальности. Аутентификация обязательна. Пользовательские данные конфиденциальны. Авторизация необязательна в рамках данной задачи.

15.2. ПРОЕКТИРОВОЧНЫЕ РЕШЕНИЯ

При обсуждении дизайна предложений и бронирования объектов размещения возникают следующие вопросы.

1. Нужно ли выполнять репликацию объектов размещения по нескольким датацентрам?
2. Как модель данных должна представлять доступность объектов размещения?

15.2.1. Репликация

Наша система Airbnb похожа на Craigslist тем, что продукты в ней имеют географическую привязку. Поиск может выполняться только по одному городу за раз. Можно воспользоваться этим фактом и закрепить хост в датацентре за городом,

в котором больше всего объявлений, или за несколькими городами с меньшим количеством объявлений. Так как производительность записи не очень важна, можно использовать репликацию с одним лидером. Чтобы минимизировать задержку чтения, вторичный лидер и последователи могут географически распределяться между датацентрами. Можно воспользоваться сервисом метаданных для хранения маппинга, связывающего города с IP-адресами хоста-лидера и хоста-последователя. Это позволит сервису найти географически близкий хост для получения объектов размещения в конкретном городе или для записи на хост-лидер, соответствующий этому городу. Такой маппинг будет иметь минимальный размер и изменяться относительно редко и только администраторами. Таким образом, можно просто реплицировать его во всех датацентрах, а администраторы будут вручную обеспечивать консистентность данных при обновлении маппинга.

Для хранения фотографий и видеороликов объектов размещения можно использовать CDN. Там же может храниться и другой статический контент, например JavaScript и CSS.

В отличие от принятой практики, можно отказаться от использования кэша в памяти. В результатах поиска выводятся только доступные объекты размещения. Если объект пользуется высоким спросом, он быстро будет зарезервирован и в результатах поиска не появится. Если объект продолжает выводиться в результатах поиска, скорее всего, он не представляет особого интереса и можно избавиться от затрат и дополнительной сложности, связанной с выделением кэша. Также можно отметить, что будет трудно обеспечить свежесть кэша, а кэшированные данные быстро устаревают.

Как обычно, все эти решения обсуждаемы, и вы должны быть способны объяснить их компромиссы.

15.2.2. Модели данных для доступности объектов размещения

Вы должны быстро предложить разные способы представления доступности объектов в модели данных и обсудить их компромиссы. На собеседовании нужно показать умение оценивать разные подходы, а не просто предложить один из них:

- *Таблица (идентификатор_объекта, дата, идентификатор_гостя)* — концептуально простое решение, но за него приходится расплачиваться необходимостью хранения нескольких строк, различающихся только датой. Например, если объект 1 забронирован пользователем 1 на весь январь, придется хранить в таблице 31 строку.
- *Таблица (идентификатор_объекта, идентификатор_гостя, заезд, выезд)* — более компактная реализация. На случай, когда гость задает поиск с датами заезда и выезда, нужен алгоритм для определения возможного перекрытия

дат. Где писать код этого алгоритма — в запросе базы данных или в бэкенде? Первый вариант сложнее обслуживать и тестировать. Но если хостам бэкенда придется загружать данные доступности объектов размещения из базы данных, это создаст лишние затраты на ввод/вывод. На собеседованиях по программированию вас могут попросить написать код обоих решений.

Существует много возможных схем баз данных.

15.2.3. Перекрытие при бронировании

Если несколько пользователей попытаются забронировать один объект с перекрывающимися датами, то заявка первого пользователя будет удовлетворена, а другие пользователи получат уведомление в интерфейсе о том, что этот объект недоступен в выбранные ими даты, и предложение найти другой доступный объект. Это может ухудшить пользовательский опыт, поэтому стоит кратко описать несколько альтернативных решений. Предложите другие возможности.

15.2.4. Рандомизация результатов поиска

Порядок результатов поиска можно рандомизировать, чтобы такие ситуации возникали реже. С другой стороны, это может помешать персонализации (например, работе рекомендательных систем).

15.2.5. Блокировка объектов в процессе бронирования

Когда пользователь кликает на результате поиска, чтобы просмотреть подробное описание объекта и, возможно, отправить заявку на бронирование, можно на несколько минут предварительно заблокировать указанные им даты для выбранного объекта. В это время другие пользователи с перекрывающимися датами в поисковых запросах не будут видеть этот объект в списке результатов. Если объект будет заблокирован после того, как другие пользователи получат свои результаты поиска, по клику на подробной информации об объекте будет выведено уведомление о блокировке и, возможно, оставшееся время блокировки на случай, если пользователь захочет повторить попытку, — нельзя исключать, что объект размещения не будет забронирован.

Это подразумевает, что часть заявок на бронирование будет потеряна. Можно решить, что избегание дублирования стоит потерянных заявок. В этом различие между Airbnb и отелями: отель может допустить избыточное бронирование дешевых номеров, поскольку ожидает вероятные отмены. Если на конкретную дату возникнет избыточное бронирование дешевых номеров, отель может предоставить гостям, которым их не хватит, более дорогие номера. У хозяев на Airbnb такой возможности нет, поэтому двойное бронирование для них недопустимо.

В разделе 2.4.2 описан механизм предотвращения конфликтов одновременных обновлений от нескольких пользователей из-за одновременного обновления общей конфигурации.

15.3. ВЫСОКОУРОВНЕВАЯ АРХИТЕКТУРА

Исходя из сказанного в предыдущем разделе, высокоуровневая архитектура может быть такой, как показана на рис. 15.1. Каждый сервис обслуживает группу взаимосвязанных функциональных требований. Это позволяет разрабатывать и масштабировать сервисы по отдельности:

- *Сервис бронирования* используется гостями для оформления брони. Этот сервис — наш непосредственный источник дохода и имеет наиболее жесткие нефункциональные требования к доступности и задержке. Более высокая задержка напрямую преобразуется в снижение дохода. Простой этого сервиса влечет самые серьезные последствия для дохода и репутации. Однако может оказаться так, что сильная консистентность менее важна и можно пожертвовать ею ради обеспечения высокой доступности и низкой задержки.
- *Сервис объявлений* используется хозяевами для создания объявлений и управления ими. Этот сервис важен, но менее критичен, чем сервис бронирования. Он работает отдельно, поскольку его функциональные и нефункциональные требования отличаются от требований для сервисов бронирования и доступности, и поэтому не должен использовать общие с ними ресурсы.
- *Сервис доступности* отслеживает доступность объявлений и используется как сервисом бронирования, так и сервисом объявлений. Требования к доступности и задержке такие же строгие, как для сервиса бронирования. Операции чтения должны быть масштабируемыми, но запись выполняется реже и не требует масштабируемости. Эта тема будет обсуждаться в разделе 15.8.
- *Сервис подтверждения* — некоторые операции (такие, как добавление новых объявлений или обновления информации в объявлениях) могут требовать подтверждения от операторов перед публикацией. Для таких сценариев можно спроектировать сервис подтверждений.
- *Сервис рекомендаций* — предоставляет гостям персонализированные рекомендации различных объявлений. Можно рассматривать его как внутренний рекламный сервис. Его подробное обсуждение выходит за рамки нашей задачи, но его можно добавить на диаграмму и упомянуть хотя бы в общих чертах.
- *Сервис нормативного соответствия* — как уже говорилось, сервисы объявлений и бронирования должны учитывать требования местного законодательства. Сервис нормативного соответствия предоставляет API сервису объявлений, так что последний предоставляет хозяевам интерфейс для создания объявлений, соответствующих требованиям местного законодательства. Сервис объявлений и сервис нормативного соответствия могут

разрабатываться разными командами, так что каждый член команды будет экспертом в отношении соответствующего сервиса. Изначально соблюдение регуляторных норм и требований выходит за рамки собеседования, но эксперта может заинтересовать, как вы планируете их обрабатывать.

- *Другие сервисы* — объединяющее название для некоторых сервисов, предназначенных для внутреннего использования (например, аналитики), которые в основном выходят за рамки собеседования.

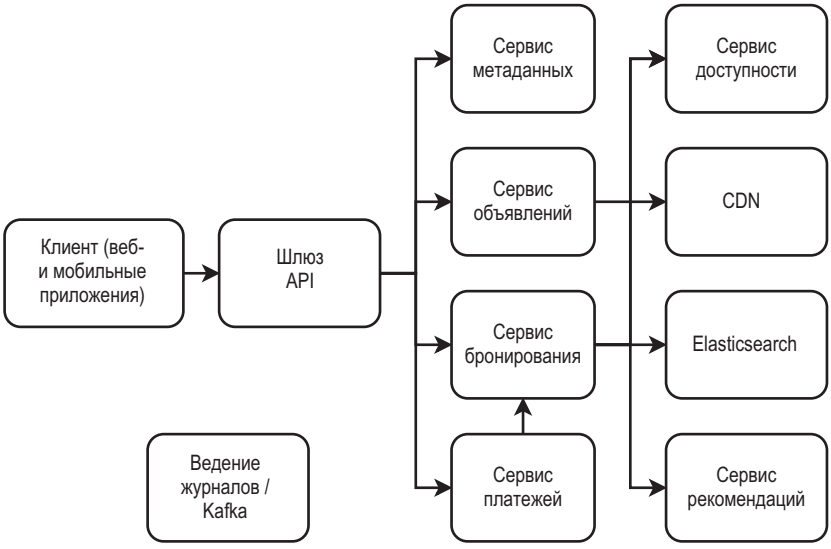

Рис. 15.1. Высокоуровневая архитектура. Как обычно, вместо шлюза API для сервисов предложений и бронирования можно использовать сервисную сеть

15.4. ФУНКЦИОНАЛЬНОЕ СЕКЦИОНИРОВАНИЕ

Можно применить функциональное секционирование по географическим регионам по аналогии с подходом, рассмотренным для Craigslist в разделе 7.9. Объявления могут размещаться в датацентре. Приложение развертывается в нескольких датацентрах, а каждый пользователь маршрутизируется к датацентру, который обслуживает его город.

15.5. СОЗДАНИЕ ИЛИ ОБНОВЛЕНИЕ ОБЪЯВЛЕНИЙ

Создание объявления можно разделить на две задачи. Первая — получение хозяином норм и требований, действующих для объявления. Вторая — отправка хозяином запроса на размещение объявления. В этой главе и операции создания, и операции обновления объявлений будут называться «запрос объявления».

На рис. 15.2 изображена диаграмма последовательности действий для получения норм и требований. Она выглядит следующим образом:

1. Хозяин находится в программе-клиенте (компоненте веб-страницы мобильного приложения), предоставляющем кнопку для создания нового объявления. Когда хозяин щелкает на кнопке, приложение отправляет запрос сервису объявлений, хранящему местонахождение пользователя. (Чтобы узнать местонахождение хозяина, можно либо вручную запросить его у самого хозяина, либо обратиться к хозяину за разрешением получить доступ к его местонахождению.)
2. Сервис объявлений направляет информацию о местонахождении сервису нормативного соответствия (см. раздел 15.10.1). Сервис нормативного соответствия возвращает ответ, включающий соответствующие нормы и требования.
3. Сервис объявлений возвращает нормы и требования клиенту. Клиент соответственно настраивает UX. Например, если существует правило, согласно которому срок бронирования должен быть не менее 14 дней, клиент выводит хозяину сообщение об ошибке, если тот установит минимальный период бронирования менее 14 дней.

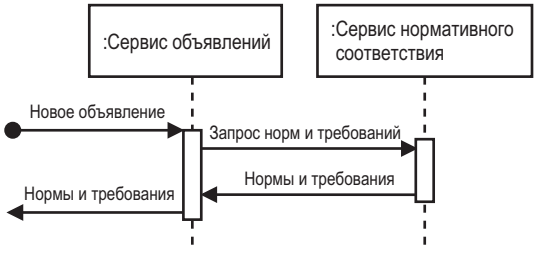

Рис. 15.2. Диаграмма последовательности получения норм и требований для предложения

На рис. 15.3 представлена диаграмма последовательности действий для упрощенного запроса объявления. Хозяин вводит информацию о своем объекте и отправляет ее. Информация передается в запросе POST сервису объявлений. Сервис объявлений:

1. Проверяет тело запроса.
2. Записывает данные в таблицу SQL для объявлений, которую мы назовем таблицей Listing. Новые объявления и определенные обновления должны подтверждаться вручную оператором. Таблица SQL Listing может содержать логический столбец с именем Approved («Одобрено»), который указывает, было ли объявление одобрено оператором.

3. Если требуется подтверждение оператора, сервис объявлений отправляет запрос POST к сервису подтверждений для уведомления операторов о необходимости проверки предложения.
4. Отправляет клиенту ответ 200.

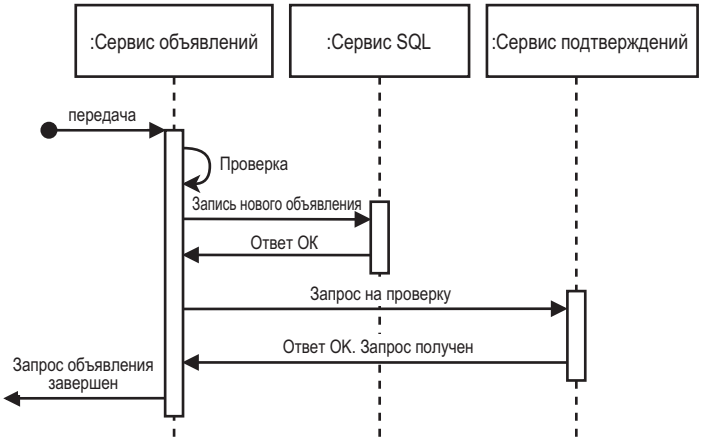

Рис. 15.3. Диаграмма последовательности выдачи упрощенного запроса на создание или обновление предложения

На рис. 15.4 шаги 2 и 3 могут выполняться параллельно с использованием CDC. Все шаги являются *идемпотентными*. Можно использовать `INSERT IGNORE` с таблицами SQL, чтобы избежать дублирования при записи (https://stackoverlow.com/a/1361368/1045085). Также можно использовать отслеживание журнала транзакций (transaction log tailing), рассмотренное в разделе 5.3.

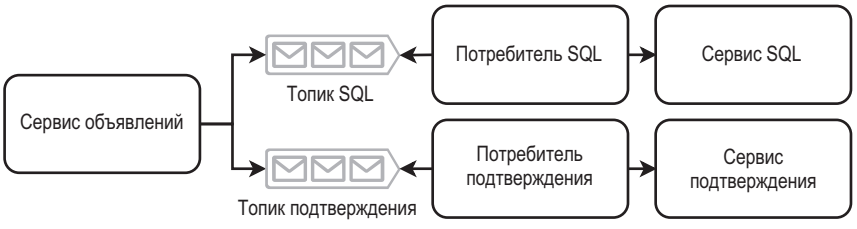

Рис. 15.4. Применение CDC для распределенных транзакций в сервисе SQL и сервисе подтверждения

На диаграмме представлен упрощенный дизайн. В реальной реализации процесс размещения объявлений может состоять из нескольких запросов к сервису объявлений. Форма для создания объявления может быть разделена на несколько частей, и хозяин может заполнять и отправлять каждую часть по отдельности,

а каждая отправка станет отдельным запросом. Например, фотографии могут включаться в объявление по одной. Хозяин может заполнить название объявления, тип и описание объекта и отправить данные в одном запросе, затем заполнить детали о цене и отправить в другом запросе, и т. д.

Другое обстоятельство, на которое стоит обратить внимание, — возможность для хозяина вносить обновления в запрос объявления, пока он ждет проверки. Каждое обновление должно применять запрос `UPDATE` к соответствующей строке таблицы.

Мы не будем подробно обсуждать уведомления, поскольку точная бизнес-логика уведомлений может быть нестандартной и часто изменяющейся. Уведомления могут быть реализованы в виде пакетного задания ETL, которое отправляет запросы к сервису, а затем дает указание общему сервису уведомлений отправить уведомления. Пакетное задание может запросить неполные предложения, а затем:

- уведомить хозяев и напомнить им, что создание объявления не завершено;
- уведомить оператора о незавершенных объявлениях, чтобы оператор связался с хозяином, напомнил ему об этом и помог завершить их создание.

15.6. СЕРВИС ПОДТВЕРЖДЕНИЙ

Эксперта может в первую очередь интересовать процесс бронирования, так что сервис подтверждений будет обсуждаться кратко.

Сервис подтверждений представляет собой внутреннее приложение с низким трафиком, которое может иметь простую архитектуру. На рис. 15.5 дизайн состоит из клиентского веб-приложения и сервиса бэкенда, отправляющего запросы к сервису объявлений и общему сервису SQL. Предполагаем, что для всех запросов необходимо ручное подтверждение, например, автоматизация всех подтверждений или отказов невозможна.

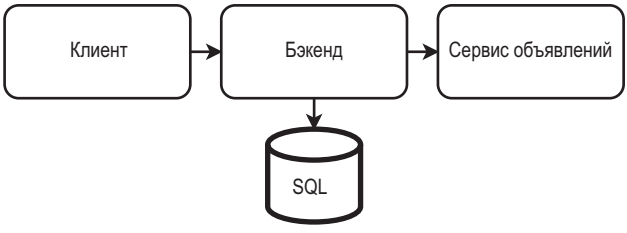

Рис. 15.5. Высокоуровневая архитектура сервиса подтверждений, в котором операторы подтверждают определенные операции (такие, как добавление или удаление объявлений)

Сервис подтверждений предоставляет конечную точку POST для запросов к сервису объявлений, требующих одобрения. Запросы можно записать в таблицу SQL с именем listing_request, которая содержит следующие столбцы:

- *id* — идентификатор (первичный ключ).
- *listing_id* — идентификатор объявления в таблице Listing в сервисе объявлений. Если бы обе таблицы находились в одном сервисе, это был бы внешний ключ.
- *created_at* — метка времени создания или обновления объявления.
- *listing_hash* — этот столбец можно включить как часть дополнительного механизма, гарантирующего, что оператор не отправит подтверждение или отказ по запросу объявления, которое изменилось во время проверки.
- *status* — перечисление со статусами запроса объявления, может принимать одно из значений: «нет», «назначено» и «проверено».
- *last_accessed* — метка времени последней загрузки запроса объявления и его возвращения оператору.
- *review_code* — перечисление (enum). Может содержать просто APPROVED (одобрено) для подтвержденных объявлений. Также возможно создание нескольких перечислений, соответствующих категориям причин для отклонения объявления: VIOLATE_LOCAL_REGULATIONS (нарушает местное законодательство), BANNED_HOST (заблокированный хозяин), ILLEGAL_CONTENT (нелегальный контент), SUSPICIOUS (подозрительное), FAIL_QUALITY_STANDARDS (не соответствует стандартам качества) и т. д.
- *reviewer_id* — идентификатор оператора, которому был назначен запрос объявления.
- *review_submitted_at* — метка времени отправки подтверждения или отказа оператором.
- *review_notes* — оператор может написать комментарий, объясняющий, почему запрос объявления подтвержден или отклонен.

Предполагая, что численность операторов составляет 10 000 и каждый оператор еженедельно проверяет до 5000 новых или обновленных объявлений, еженедельно в таблицу SQL будет записываться 50 миллионов строк. Если каждая строка занимает 1 Кбайт, таблица подтверждений вырастет на 1 Кбайт × 50M × 30 дней = 1,5 Тбайт за месяц. Можно хранить данные за 1–2 месяца в таблице SQL и выполнять периодическое пакетное задание для архивации старых данных в хранилище объектов.

Также можно спроектировать конечные точки и таблицу SQL для каждого оператора. Оператор сначала отправляет запрос GET, содержащий его идентификатор,

для загрузки запроса объявления из таблицы listing_request. Чтобы один запрос объявления не был назначен нескольким операторам, бэкенд запускает транзакцию SQL:

1. Если оператору уже назначен запрос объявления, вернуть этот назначенный запрос, выбрать (SELECT) строку со статусом assigned и идентификатором оператора в reviewer_id.
2. Если запрос объявления не назначен, выбрать (SELECT) строку с минимальной меткой времени created_at со статусом none. Это и будет назначенный запрос объявления.
3. Обновить (UPDATE) статус значением assigned, а reviewer_id — идентификатором оператора.

Бэкенд возвращает этот запрос объявления оператору, который проверяет и подтверждает или отклоняет его. На рис. 15.6 представлена диаграмма последовательности синхронного подтверждения. Подтверждение или отклонение выполняется запросом POST к таблице Approval, который запускает следующие действия:

1. Обновить (UPDATE) строку в таблице listing_request. Обновить столбцы status, review_code, review_submitted_at и review_notes.

   Возникает возможная ситуация гонки, в которой хозяин может обновить запрос объявления, пока оператор его проверяет. Следовательно, запрос POST должен содержать хеш объявления, который был ранее возвращен сервисом подтверждения оператору, а бэкенд должен проверить, что этот хеш идентичен текущему. Если хеши различны, вернуть обновленный запрос объявления оператору, который должен будет повторить проверку.

   Можно попытаться выявить эту ситуацию гонки, проверяя, что метка listing_request.last_accessed относится к более позднему времени, чем listing_request.review_submitted_at. Однако этот способ ненадежен, потому что часы разных хостов, показания которых хранятся в столбце «метки времени», не могут быть синхронизированы идеально. Кроме того, время может изменяться по разным причинам: летнее время, перезапуск сервиса, периодическая синхронизация часов сервера по эталонному серверу и т. д. В распределенных системах невозможно обеспечивать согласованность на основании показаний часов (Мартин Клеппман, «Designing Data-Intensive Applications»).
2. Отправить запрос PUT сервису объявлений. Запрос обновляет (UPDATE) столбцы listing_request.status и listing_request.reviewed_at. И снова сначала следует получить хеш и убедиться, что он не отличается от отправленного. Оба запроса SQL заключаются в транзакцию.
3. Отправить запрос POST сервису бронирования, чтобы сервис бронирования вывел это предложение для гостей. Альтернативное решение представлено на рис. 15.7.

Часы Лэмпорта и векторные часы

Часы Лэмпорта (https://martinfowler.com/articles/patterns-of-distributed-systems/lamport-clock.html) — метод упорядочения событий в распределенной системе. Векторные часы — другой, более сложный метод. Подробности см. в главе 11 книги Джорджа Кулуриса (George Coulouris), Джин Доллимор (Jean Dollimore), Тима Киндберга (Tim Kindberg) и Гордона Блэра (Gordon Blair) «Distributed Systems: Concepts and Design», Pearson, 2011.

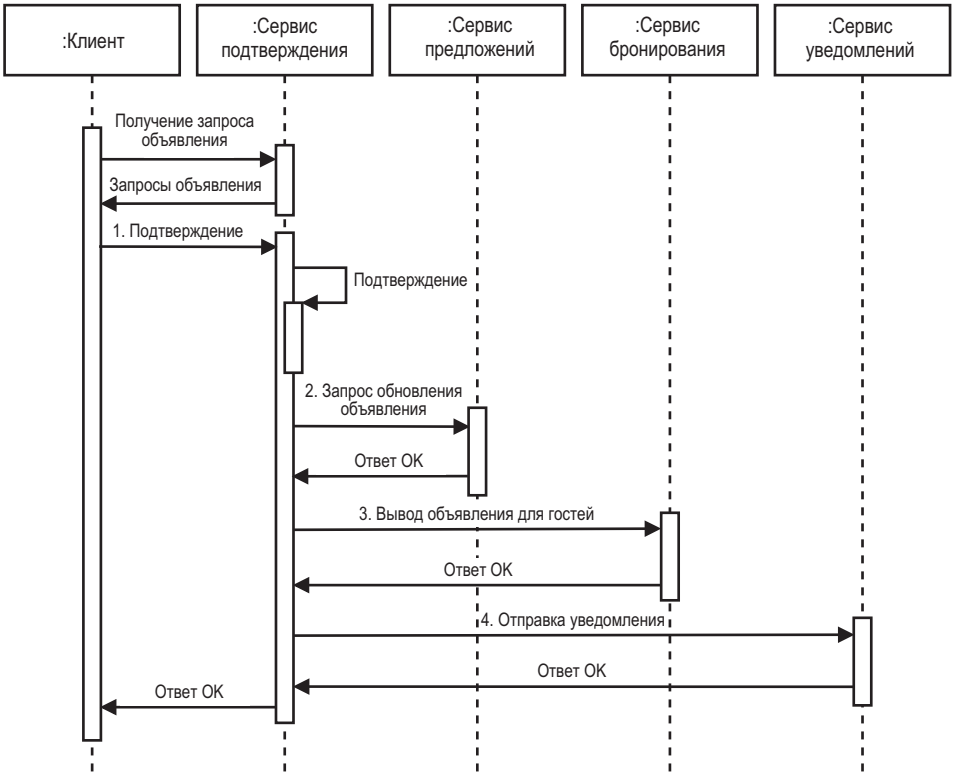

Рис. 15.6. Диаграмма последовательности действий выборки запросов объявления, за которой следует синхронное подтверждение запроса объявления. Сервис подтверждений может быть оркестратором саги

4. Бэкенд также обращается с запросом к общему сервису уведомлений (глава 9), чтобы известить хозяина о подтверждении или отказе.
5. Наконец, бэкенд отправляет клиенту ответ 200. Эти шаги должны быть запрограммированы идемпотентно, чтобы любые из них можно было повторить в случае отказа хоста.

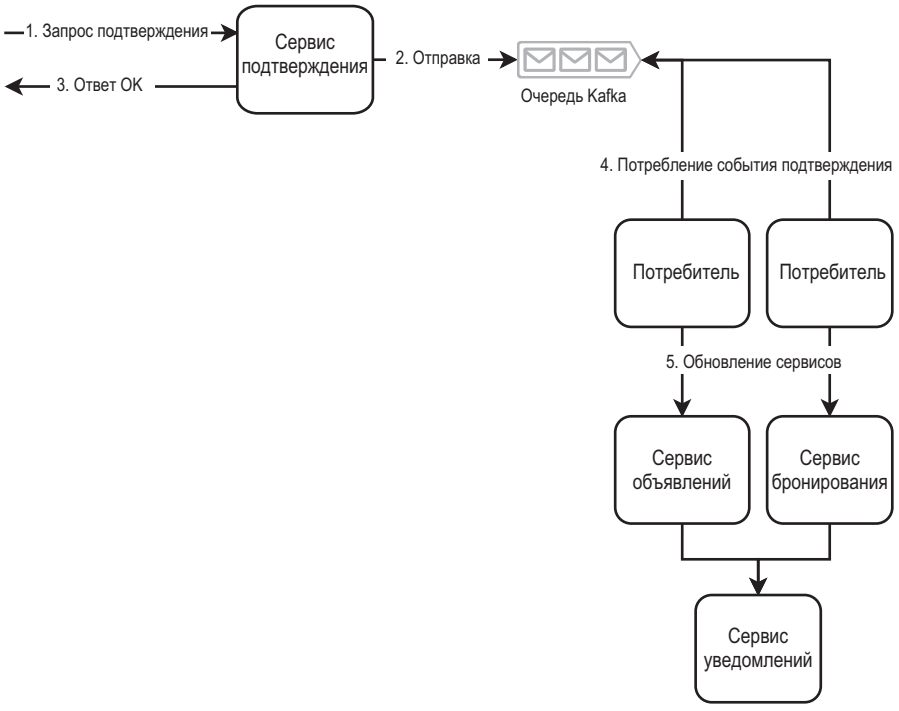

Рис. 15.7. Асинхронное решение подтверждения запроса объявления на базе CDC. Так как все запросы допускают повторную попытку, использовать сагу не нужно

Обсудите, как запрос POST может быть идемпотентным, на случай, если до завершения всех шагов произойдет сбой и запрос нужно будет повторить. Например:

- Перед запросом на уведомление бэкенд может отправить сервису уведомлений команду проверить, был ли выдан конкретный запрос на уведомление.
- Чтобы избежать дублирования строк в таблице Approval, при вставке строки SQL можно использовать инструкцию `IF NOT EXISTS`.

Как видите, синхронный запрос включает запросы к нескольким сервисам и может приводить к большой задержке. Неудачный запрос к любому сервису создаст несогласованность.

Но нельзя ли использовать CDC (Change Data Capture)? На рис. 15.7 представлено такое асинхронное решение. По запросу на подтверждение сервис подтверждений отправляет событие в очередь Kafka и возвращает ответ 200. Потребитель потребляет из очереди Kafka и отправляет запросы ко всем остальным сервисам. Частота подтверждений низка, так что потребитель может применить экспоненциальную задержку и повторить попытку, чтобы избежать

быстрого опроса очереди Kafka, когда она пуста, и проводить опрос только один раз в минуту при пустой очереди.

Сервис уведомлений уведомляет хозяина только после обновления сервисов объявлений и бронирования, поэтому он потребляет из двух топиков Kafka, каждый из которых соответствует одному сервису. Когда сервис уведомлений потребляет из одного топика событие, соответствующее конкретному событию подтверждения объявления, ему придется ожидать события от другого сервиса, соответствующего тому же событию подтверждения, после чего он сможет отправить уведомление. Таким образом, сервису уведомлений понадобится база данных для хранения этих событий. Эта база данных не показана на рис. 15.7.

В качестве дополнительной меры защиты от скрытых ошибок, которые могут вызвать несогласованность между сервисами, можно реализовать пакетное задание ETL для аудита трех сервисов. Это задание будет оповещать разработчиков в случае обнаружения несогласованности.

Для этого процесса мы используем CDC, а не сагу, поскольку не ожидаем, что какой-то из сервисов отклонит запрос, и поэтому компенсирующие транзакции не потребуются. У сервиса объявлений и сервиса бронирования нет причин запрещать публикацию объявлений, а у сервиса уведомлений нет причин не отправлять пользователю уведомление.

Но что, если пользователь удалит свою учетную запись до того, как его объявление будет подтверждено? Нам понадобится процесс CDC для деактивации или удаления объявлений и отправки запросов к другим сервисам по мере необходимости. Если сервисы, задействованные в процессе утверждения на рис. 15.6, получат запрос пользователя на удаление непосредственно перед запросом подтверждения, они могут либо сохранить информацию о том, что объявление недействительно, либо удалить объявление. Тогда запрос подтверждения не активирует объявление. Обсудите с экспертом компромиссы разных решений и другие значимые детали, которые придут вам на ум. Ваше внимание к ним будет оценено по достоинству.

Требуемый функционал может различаться. Например, в подтверждении объявления могут участвовать сразу несколько операторов. Упомяните эту возможность и обсудите ее, если эксперт сочтет ее интересной.

Операторы могут специализироваться на проверке запросов объявления, относящихся к определенным юрисдикциям. Как назначить оператору подходящие объявления? Ваше приложение уже функционально разделено по географическим областям, так что оператор может проверять объявления из конкретного датацентра, ничего другого описанная архитектура и не требует. Также предложите следующие возможности:

- Запрос JOIN между таблицами listing_request и listing для получения объявлений, относящихся к конкретной стране или городу. Таблицы listing_

request и listing принадлежат разным сервисам, поэтому понадобится другое решение:

- Изменение дизайна системы. Сервисы объявлений и подтверждений объединяются, так что обе таблицы будут принадлежать одному сервису.
- Реализация логики соединения на уровне приложения, что имеет такие недостатки, как затраты ввода/вывода на передачу данных между сервисами.
- Денормализация или дублирование данных объявлений с включением столбца location в таблицу listing_request либо дублирование таблицы listing в сервисе подтверждений. Физическое местоположение объявления при этом не меняется, поэтому возникает низкий риск несогласованности, вызванный денормализацией или дублированием. Впрочем, несогласованность все же возможна из-за ошибок в коде или из-за того, что местоположение было введено неверно, а затем исправлено.

- Идентификатор объявления может содержать идентификатор города, так что город объявления можно определить по идентификатору объявления. Ваша компания может вести список (*идентификатор*, *город*), к которому может обратиться любой сервис. Этот список должен быть доступен только для добавления данных, так что затратные и рискованные миграции данных выполнять не придется.

Как уже говорилось, подтвержденные объявления будут копироваться в сервис бронирования. Так как сервис бронирования может получать высокий трафик, частота сбоев на этом шаге может быть максимальной. Как во многих представленных решениях, можно реализовать экспоненциальную задержку с повтором или очередь недоставленных сообщений. Трафик от сервиса подтверждений к сервису бронирования пренебрежимо мал по сравнению с трафиком от гостей, так что мы попытаемся сократить вероятность простоев сервиса бронирования за счет сокращения трафика от сервиса подтверждения.

Наконец, можно обсудить автоматизацию некоторых подтверждений или отказов. Правила можно определить в таблице SQL с именем Rules; функция будет загружать эти правила и применять их к содержимому объявления. Также можно воспользоваться средствами машинного обучения. Модели МО тренируются в сервисе МО, после чего идентификаторы отобранных моделей помещаются в таблицу Rules. Таким образом, функция может отправлять содержимое объявлений с идентификаторами моделей сервису МО, который возвращает подтверждение, отказ или неопределенный результат (то есть необходимость ручной проверки). Столбец listing_request.reviewer_id может содержать значение вида AUTOMATED (автоматизированная проверка), а столбец listing_request.review_code — значение INCONCLUSIVE для неопределенного результата проверки.

15.7. СЕРВИС БРОНИРОВАНИЯ

Упрощенный процесс бронирования/резервирования включает следующие шаги:

1. Гость отправляет запрос на поиск объявлений, соответствующих указанным ниже параметрам, и получает список доступных объявлений. Каждое объявление в списке результатов может содержать миниатюру и краткую информацию. Как было сказано выше, в разделе требований, другие подробности в нашем примере не учитываются. Параметры:
   - город;
   - дата заезда;
   - дата выезда.
2. Гость фильтрует результаты по цене и другим критериям.
3. Гость кликает на объявлении, чтобы просмотреть более подробную информацию, включая фотографии в высоком разрешении и видео (если есть). Отсюда гость может вернуться к списку результатов.
4. Гость выбирает объявление, отправляет заявку на бронирование и получает подтверждение или ошибку.
5. Если гость получает подтверждение, он переходит к оплате.
6. Гость может передумать и отправить запрос на отмену.

По аналогии с сервисом объявлений, рассмотренным ранее, можно отправлять уведомления в некоторых случаях:

- Уведомление гостей и хозяев после успешного завершения или отмены бронирования.
- Если гость ввел подробную информацию в заявке на бронирование, но не завершил заявку, напомнить ему через несколько часов или дней о том, что ее нужно завершить.
- Рекомендации объявлений гостям на основании разных параметров: прошлых заявок, просмотренных объявлений, прочей активности в интернете, демографических данных и т. д. Объявления могут подбираться рекомендательной системой.
- Уведомления, связанные с платежами. Вы можете выбрать вариант с внесением условного депозита до того, как хозяин примет заявку, или запрашивать платеж только после принятия заявки хозяином. Логика уведомлений будет изменяться соответствующим образом.

Кратко обсудим требования к масштабированию. Как уже говорилось, можно провести функциональное секционирование предложений по городам. Допустим,

в городе поддерживается до 1 миллиона предложений. Можно сделать явно завышенную оценку до 10 миллионов ежедневных запросов поиска, фильтрации и подробной информации по объявлениям. Даже если предположить, что эти 10 миллионов запросов выполняются за 1 час, получается менее 3000 запросов в секунду; с такой нагрузкой может справиться один хост или небольшая группа хостов. Тем не менее архитектура, рассмотренная в этом разделе, способна справиться с намного бóльшим трафиком.

На рис. 15.8 изображена высокоуровневая архитектура сервиса бронирования. Все запросы обрабатываются сервисом бэкенда, который обращается к общим сервисам Elasticsearch или SQL.

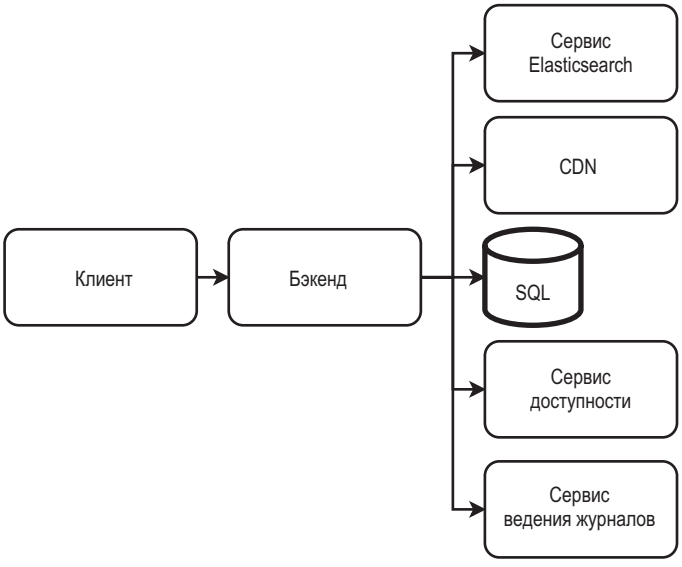

Рис. 15.8. Высокоуровневая архитектура сервиса бронирования

Запросы поиска и фильтрации обрабатываются сервисом Elasticsearch. Сервис Elasticsearch также может осуществлять разбиение на страницы (https://www.elastic.co/guide/en/elasticsearch/reference/current/paginate-search-results.html), чтобы сэкономить память и нагрузку на процессор за счет возвращения небольшого количества результатов за раз. Elasticsearch поддерживает нечеткий поиск, особенно полезный для гостей, допускающих ошибки в названиях и адресах.

Запрос подробной информации по объявлению форматируется в запрос SQL с использованием ORM и направляется сервису SQL. Фотографии и видео загружаются из CDN. Запрос на бронирование направляется сервису доступности, который подробно рассматривается в следующем разделе. Операции записи в базу данных SQL-сервиса выполняются:

1. Запросами на бронирование.
2. Сервисом подтверждений, о чем было сказано в предыдущем разделе. Сервис подтверждений выполняет нечастые обновления информации по объявлениям.
3. Запросами на отмену бронирования, в результате чего объявления снова становятся доступными. Это происходит в случае неудачной оплаты.

Сервис SQL, используемый сервисом бронирования, может иметь архитектуру «лидер — последователь», рассмотренную в разделе 4.3.2. Нечастые операции записи выполняются на хосте лидера и реплицируются на хостах последователей. Сервис SQL может содержать таблицу Booking со следующими столбцами:

- *id* — (первичный ключ) идентификатор, присваиваемый бронированию.
- *listing_id* — идентификатор объявления, назначаемый сервисом объявлений. Если бы таблица находилась в сервисе объявлений, этот столбец был бы внешним ключом.
- *guest_id* — идентификатор гостя, оформившего бронь.
- *check_in* — дата заезда.
- *check_out* — дата выезда.
- *timestamp* — время вставки или обновления строки. Этот столбец используется только для учетных целей.

Другие операции записи в этом процессе обращены к сервису доступности:

1. Запрос бронирования или отмены изменяет доступность объявления в соответствующие даты.
2. Можно рассмотреть возможность блокирования объявлений на 5 минут на шаге 3 процесса бронирования (запрос дополнительной информации по объявлению), потому что гость может отправить заявку на бронирование. Это означает, что объявление не будет выводиться в результатах поиска для других гостей, отправляющих поисковые запросы на даты, перекрывающиеся с датами в текущем запросе. И наоборот, можно разблокировать объявление быстрее (до истечения 5 минут), если гость отправляет запрос поиска или фильтрации, который означает, что он вряд ли будет бронировать этот объект.

Индекс Elasticsearch должен обновляться при изменении доступности или информации по объявлению. Добавление или обновление объявления требует запросов записи как к сервису SQL, так и к сервису Elasticsearch. Как упоминалось в главе 5, эта схема может быть реализована в виде распределенной транзакции для предотвращения несогласованности, если во время записи в какой-либо из сервисов произойдет сбой. Запрос бронирования требует записи в сервисы SQL как в сервисе бронирования, так и в сервисе доступности (см. следующий раздел) и также должен быть оформлен в виде распределенной транзакции.

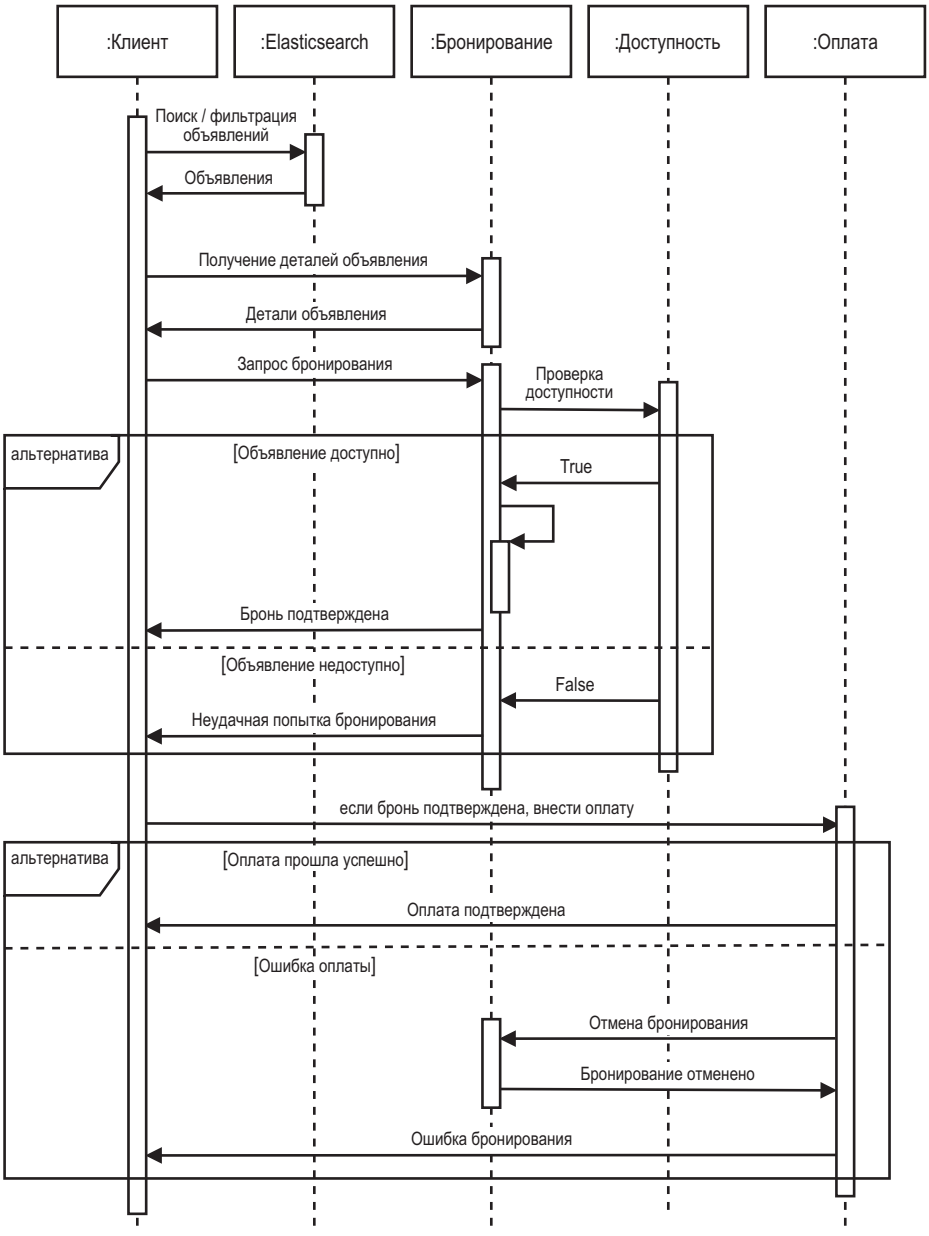

Рис. 15.9. Диаграмма последовательности действий упрощенного процесса бронирования. Многие подробности опущены. Например, в получении подробного описания объявления может участвовать CDN. Хозяева не могут вручную подтверждать или отклонять заявки на бронирование. Проведение оплаты требует большого количества запросов к нескольким сервисам. Запросы к сервису уведомлений не показаны

Если бронирование приводит к тому, что объявление больше не соответствует требованиям к публикации, сервис бронирования должен обновить свою базу данных, чтобы бронирование этого объявления больше не осуществлялось, а также обновить сервис Elasticsearch, чтобы это объявление перестало появляться в результатах поиска.

Результат Elasticsearch может сортировать объявления по убыванию рейтинга объекта (по оценкам гостей). Результаты также могут сортироваться сервисом машинного обучения. Эти возможности мы рассматривать не будем.

На рис. 15.9 представлена последовательная диаграмма упрощенного процесса бронирования.

Наконец, можно предположить, что гостей будет много и они будут проводить поиск по многим объявлениям и просматривать описание перед тем, как отправить заявку на бронирование. Следовательно, можно выделить функции поиска и просмотра и функцию бронирования в отдельные сервисы, чтобы их можно было масштабировать по отдельности. Сервис поиска и просмотра объявлений будет получать больше трафика, и ему будет выделяться больше ресурсов, чем сервису отправки заявок на бронирование.

15.8. СЕРВИС ДОСТУПНОСТИ

Сервис доступности должен предотвращать следующие сценарии:

- Двойное бронирование.
- Хозяин не видит бронирование от гостя.
- Хозяин помечает некоторые даты как недоступные, но гость видит эти даты.
- Отдел технической поддержки перегружен жалобами гостей и хозяев на эти недостатки.

Сервис доступности предоставляет следующие конечные точки:

- Для идентификатора местоположения, идентификатора типа объявления, даты заезда и даты выезда получить (GET) доступные объявления.
- Блокировка объявлений в указанный период (от даты заезда до даты выезда) на несколько (например, пять) минут.
- CRUD-операции с бронированием, от указанной даты заезда до указанной даты выезда.

На рис. 15.10 представлена высокоуровневая архитектура сервиса доступности. Он состоит из сервиса бэкенда, который отправляет запросы к общему сервису SQL. Общий сервис SQL использует архитектуру «лидер/последователь», изображенную на рис. 4.1 и 4.2.

Рис. 15.10. Высокоуровневая архитектура сервиса доступности

Сервис SQL может содержать таблицу доступности со следующими столбцами (таблица не имеет первичного ключа):

- *listing_id* — идентификатор объявления, назначенный сервисом объявлений.
- *date* — дата доступности.
- *booking_id* — идентификатор бронирования/резерва, назначенный сервисом бронирования при установлении брони.
- *available* — строковое поле, которое является, по сути, перечислением (enum). Оно сообщает состояние доступности объявления — доступно, заблокировано, забронировано. Для экономии места можно удалить это поле, если комбинация (listing_id, date) не заблокирована или забронирована. Однако нам нужна высокая загруженность объекта размещения, так что экономия места будет незначительной. Другой недостаток заключается в том, что сервис SQL должен выделить достаточный объем хранилища для всех возможных строк, так что если мы экономим место, не вставляя строки без абсолютной на то необходимости, можно не увидеть, что имеющегося места недостаточно, пока мы не получим высокую загруженность.
- *timestamp* — время вставки или обновления строки.

Процесс блокировки объявлений рассматривался в предыдущем разделе. Можно вывести в клиенте (веб- или мобильном приложении) шестиминутный таймер. Таймер на стороне клиента должен длиться чуть дольше, чем таймер бэкенда, потому что часы клиента и хоста бэкенда не могут быть идеально синхронизированы.

Такой механизм блокировки предложений может сократить (но не предотвратить полностью) риск создания несколькими гостями перекрывающихся заявок на бронирование. Для предотвращения перекрытий бронирования можно воспользоваться блокировкой строк SQL. (См. https://dev.mysql.com/doc/refman/8.0/en/glossary.html#glos_exclusive_lock и https://www.postgresql.org/docs/current/explicit-locking.html#LOCKING-ROWS.) Сервис бэкенда должен использовать транзакцию SQL на хосте-лидере. Сначала отправьте запрос `SELECT`, чтобы проверить доступность объявления в запрашиваемый период. Затем отправьте запрос `INSERT` или `UPDATE`, чтобы пометить объявление соответствующим образом.

Компромисс консистентности в архитектуре SQL «лидер — последователь» состоит в том, что результаты поиска могут содержать недоступные объявления. Если гость попытается забронировать недоступное объявление, сервис бронирования

вернет ответ 409. Это не должно сильно испортить впечатление от работы с приложением, потому что пользователь может ожидать, что объявление может быть кем-то забронировано, пока он его просматривает. Однако стоит добавить метрику в сервис мониторинга, чтобы отслеживать такие ситуации; тогда вы получите оповещение и сможете отреагировать, если проблема будет возникать слишком часто.

Ранее в этой главе мы обсудили, почему мы не кэшируем популярные пары (listing, date). Если вы решите это сделать, можно реализовать стратегию кэширования, подходящую для нагрузок с интенсивным чтением; эта тема обсуждается в разделе 4.8.1.

Сколько места для этого понадобится? Если каждый столбец занимает 64 бита, строка будет занимать 40 байт. Один миллион предложений будет занимать 7,2 Гбайт за 180 дней, что легко реализуемо на одном хосте. При необходимости старые данные можно удалять вручную, чтобы освободить место.

Альтернативная схема таблицы SQL может напоминать таблицу Booking, рассмотренную в предыдущем разделе, кроме того, что в ней может присутствовать столбец с именем status или availability. Он сообщает, было ли объявление заблокировано или забронировано. Алгоритм поиска объявлений, доступных между заданной датой заезда и датой выезда, скорее относится к области программирования. Вам могут предложить реализовать его на собеседовании по программированию, но не на собеседовании по проектированию систем.

15.9. ВЕДЕНИЕ ЖУРНАЛОВ, МОНИТОРИНГ И ОПОВЕЩЕНИЯ

Кроме того, что рассматривалось в разделе 2.5 (процессор, память, использование диска в Redis, использование диска в Elasticsearch), следует отслеживать и отправлять оповещения для ряда других событий.

Предусмотрите оповещения в случае необычно высокой частоты бронирования, создания объявлений или отмен. Другой пример — необычно высокая частота объявлений, которые вручную или программно помечаются как имеющие дефекты.

Определите сквозные пользовательские истории, например последовательность действий, выполняемых хозяином при создании объявления или гостем при оформлении брони. Отслеживайте соотношение завершенных и незавершенных пользовательских историй/потоков, создайте оповещения для необычно высокой доли ситуаций, в которых пользователи не проходят через всю историю/поток.

Также можно определять и отслеживать частоту нежелательных пользовательских историй, например заявок на бронирование, которые так и не были завершены или были отменены после общения гостя и хозяина.

15.10. ДРУГИЕ ВОЗМОЖНЫЕ ТЕМЫ ОБСУЖДЕНИЯ

Может показаться, что сервисы и бизнес-логика, обсуждаемые в этой главе, показаны поверхностно и излишне упрощены. На собеседовании вы можете проектировать дополнительные сервисы и обсудить их требования, пользователей и межсервисные взаимодействия. Также можно более подробно проанализировать пользовательские истории и соответствующие им особенности дизайна системы:

- Пользователя могут интересовать объявления, которые не совсем точно соответствуют критериям поиска. Например, дата заезда и/или выезда может немного отличаться от введенных или объявления в соседних городах тоже могут оказаться приемлемыми. Как же спроектировать сервис поиска, который возвращает такие результаты? Изменить ли поисковый запрос перед тем, как передавать его Elasticsearch, или же спроектировать индекс Elasticsearch, который рассматривает такие результаты как релевантные?
- Какую еще функциональность можно добавить для хозяев, гостей, операторов и других пользователей? Например, можно ли спроектировать систему, при помощи которой гости смогут сообщать о недопустимых объявлениях? Можно ли спроектировать систему, которая отслеживает поведение хозяев и гостей и рекомендует возможные штрафные санкции, например ограничения на использование сервиса или блокировку учетной записи?
- Функциональные требования, определенные ранее как не входящие в контекст собеседования, и особенности их архитектуры (например, обеспечивать ли требования в имеющихся сервисах или создать отдельные сервисы).
- Мы не обсуждали тему поиска. Подумайте, стоит ли предоставить гостям возможность искать объявления по ключевым словам, для чего объявления придется проиндексировать. Используйте Elasticsearch или спроектируйте собственный сервис поиска.
- Расширьте ассортимент объявлений (например, объявления для бизнес-путешественников).
- Разрешенное двойное бронирование (по аналогии с отелями). Если объект размещения оказывается недоступен, гостю предоставляется более дорогой вариант, так как более дорогие объекты обычно менее популярны.
- Пример аналитической системы рассматривается в главе 17.
- Вывод статистики для пользователей (например, популярность объявлений).
- Персонализация (например, рекомендательная система для объявлений). Так, рекомендательный сервис может продвигать новые объявления, чтобы у них быстро появились гости; это может быть полезно для новых хозяев.
- На собеседованиях на вакансии фронтенд-разработчика или UX-дизайнера также могут обсуждаться рабочие процессы UX.
- Защита от мошенничества и снижение рисков.

15.10.1. Работа с нормативными требованиями

Можно рассмотреть возможность проектирования и реализации специального сервиса нормативного соответствия, чтобы предоставить стандартный API для работы с нормативными требованиями. Все остальные сервисы должны проектироваться с учетом взаимодействия с этим API, поэтому они должны быть гибкими, чтобы адаптироваться к изменениям законодательных норм, или по крайней мере допускать простое изменение дизайна в ответ на непредвиденные изменения.

По опыту автора, многие компании не уделяют внимания проектированию сервисов, обладающих необходимой гибкостью с учетом изменения законодательных норм. Из-за этого им приходится тратить значительные ресурсы на переработку архитектуры, изменение реализации и миграцию при каждом изменении нормативных требований.

Упражнение

Обсудите различия в нормативных требованиях для Airbnb и Craigslist.

Законы о конфиденциальности данных актуальны для многих компаний. Примеры — COPPA (https://www.ftc.gov/enforcement/rules/rulemaking-regulatory-reform-proceedings/childrens-online-privacy-protection-rule), GDPR (https://gdpr-info.eu/) и CCPA (https://oag.ca.gov/privacy/ccpa). Некоторые правительства требуют, чтобы компании делились информацией о событиях, происходящих в их юрисдикции, или чтобы данные граждан не хранились за пределами страны.

Нормативные требования могут влиять на основную сферу деятельности компании. В случае Airbnb существуют требования, распространяющиеся напрямую на хозяев и гостей. Примеры:

- Максимальная продолжительность размещения объявления ограничена заданным количеством дней в году.
- Объектом размещения может выступать только недвижимость, построенная до или после определенного года.
- Бронирование не может включать некоторые даты (например, государственные праздники).
- В некоторых городах устанавливается минимальная или максимальная продолжительность бронирования.
- В некоторых городах или районах объявления могут быть полностью запрещены.
- В объявлении необходимо указать на наличие системы противопожарной безопасности: датчиков угарного газа, дыма, указателей пожарных выходов.
- Могут действовать другие нормы пригодности для жилья и безопасности.

В пределах страны определенные нормативные требования могут относиться к объявлениям, удовлетворяющим определенным условиям, причем конкретика может зависеть от страны, штата, города или даже адреса (например, некоторые жилые комплексы могут устанавливать собственные правила).

ИТОГИ

- Airbnb — приложение для бронирования, маркетплейс, а также инструмент поддержки пользователей и операций. Основные группы пользователей — хозяева, гости и группа эксплуатации.
- Продукты Airbnb имеют географическую привязку, так что объявления могут группироваться по географическому расположению датацентров.
- Большое количество сервисов, задействованных в создании объявлений и бронирований, в ходе собеседования полностью обсудить невозможно. Можно только перечислить основные сервисы и их функциональность.
- Создание объявления может потребовать нескольких запросов от хоста Airbnb, обеспечивающих соответствие объявления местным нормативным требованиям.
- Отправленный хозяином запрос на создание объявления может требовать ручной проверки администратором/оператором. После проверки объявление будет выводиться в результатах поиска и станет доступно для бронирования гостями.
- Взаимодействие между сервисами должно быть асинхронным, если низкая задержка не обязательна. Для обеспечения асинхронных взаимодействий применяются средства распределенных транзакций.
- Кэширование не всегда уместно для сокращения задержки, особенно если кэш быстро устаревает.
- При проектировании сложных транзакций чрезвычайно полезно использовать диаграммы архитектуры и последовательности действий.

16 Проектирование ленты новостей

В ЭТОЙ ГЛАВЕ

- ✓ Проектирование персонализированной масштабируемой системы
- ✓ Фильтрация элементов ленты новостей
- ✓ Проектирование ленты новостей с поддержкой графики и текста

Задача — спроектировать ленту новостей, которая предоставляет пользователю список элементов, отсортированных приблизительно в обратном хронологическом порядке и соответствующих темам, выбранным пользователем. Один элемент новостей может относиться к одной-трем темам. В любой момент пользователь может выбрать до трех тем.

Этот вопрос часто встречается на собеседованиях по проектированию систем. В этой главе термины «новостной элемент» и «пост» будут использоваться как синонимы. В таких приложениях социальных сетей, как Facebook или Twitter, лента новостей пользователя обычно содержит посты друзей/знакомых. Однако в нашей ленте новостей пользователи получают посты любых людей, а не только их друзей.

16.1. ТРЕБОВАНИЯ

Ниже перечислены функциональные требования системы ленты новостей, которые, как обычно, можно обсудить с экспертом примерно за 5 минут в режиме «вопрос — ответ».

- Пользователь может выбирать темы, представляющие для него интерес. Система поддерживает до 100 тегов (мы будем использовать термин «тег» вместо «новостная тема»).
- Пользователь может загрузить список новостей на английском языке, всего до 1000 элементов. За раз загружаются 10 элементов.
- Хотя пользователь может загрузить до 1000 элементов, система должна архивировать все элементы.
- В исходной версии пользователи будут получать одинаковые элементы независимо от их географического положения. Затем мы рассмотрим возможность персонализации в зависимости от таких факторов, как местонахождение и язык.
- Сначала выводятся последние новости; иначе говоря, новостные элементы должны быть упорядочены в обратном хронологическом порядке, но упорядочение может быть приблизительным.
- Компоненты новостного элемента:
  - Новый элемент обычно содержит несколько текстовых полей, например заголовок максимальной длины 150 символов и тело максимальной длины 10 тысяч символов. Для простоты ограничимся одним текстовым полем максимальной длины 10 тысяч символов.
  - Метка времени UNIX, обозначающая время создания элемента.
  - В исходной версии не рассматривается использование аудио, изображений и видео. Если хватит времени, можно рассмотреть возможность включения 0–10 файлов изображений размером до 1 Мбайт каждое.

  СОВЕТ Исходные функциональные требования не включают изображения, поскольку изображения существенно усложняют проектирование системы. Сначала можно спроектировать систему, которая работает только с текстом, а затем подумать, как ее расширить для поддержки графики и других мультимедиа-материалов.

  - Некоторые элементы не должны выводиться в ленте, так как содержат нежелательный контент.

Перечисленные ниже пункты в основном или полностью выходят за рамки функциональных требований:

- Версионирование не рассматривается, поскольку статьи могут существовать в нескольких версиях. Автор может добавить в статью текст или элемент мультимедиа либо отредактировать статью для исправления ошибок.
- В исходной версии не рассматривается аналитика пользовательских данных (например, темы, которыми пользователь интересовался, выводимые пользователю статьи и статьи, которые он прочитал). Сложные рекомендательные системы также не рассматриваются.
- Не понадобятся другие средства персонализации или инструменты социальных сетей (такие, как «поделиться» или «комментировать»).
- Источники новостных элементов не рассматриваются. Достаточно предоставить конечную точку POST API для добавления новостей.
- В исходной версии поиск рассматривать не нужно. Поиском можно заняться после выполнения других требований.
- Не нужны средства монетизации, такие как регистрация пользователя, платежи или подписки. Предполагается, что все статьи бесплатны. Также не нужно рассматривать включение рекламы в статьи.

Нефункциональные требования системы ленты новостей могут выглядеть так:

- Масштабируемость до поддержки 100 тысяч ежедневных активных пользователей, каждый из которых в среднем отправляет 10 запросов в день, и до 1 миллиона новостных элементов в день.
- Для чтения необходима высокая производительность с односекундной задержкой 99-го процентиля.
- Конфиденциальность пользовательских данных.
- Консистентность в конечном счете с задержкой до нескольких часов допустима. Пользователям не обязательно иметь возможность просматривать статьи сразу же после их публикации, но желательна задержка не более нескольких секунд. В некоторых новостных приложениях существует требование возможности назначить выбранным новостям статус «срочная» и доставлять их немедленно с высоким приоритетом, но наша лента новостей такую возможность не поддерживает.
- Для записи необходима высокая доступность. Высокая доступность для чтения — полезное, но необязательное свойство, так как пользователи могут кэшировать старые новости на своих устройствах.

16.2. ВЫСОКОУРОВНЕВАЯ АРХИТЕКТУРА

Начнем с высокоуровневой архитектуры ленты новостей, представленной на рис. 16.1. Источники новостных элементов отправляют новости сервису

поглощения данных на бэкенде, и он записывает их в базу данных. Пользователи обращаются с запросами к сервису ленты новостей, который получает новостные элементы из БД и возвращает их пользователям.

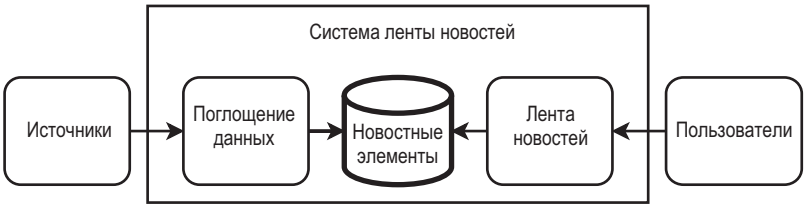

Рис. 16.1. Исходная высокоуровневая архитектура ленты новостей. Источники новостей отправляют элементы сервису поглощения данных, который обрабатывает их и сохраняет в базе данных. Пользователи обращаются с запросами к сервису ленты новостей, который получает новые элементы из БД

Несколько замечаний по поводу этой архитектуры:

- Сервис поглощения данных должен обладать высокой доступностью и обрабатывать высокий и непредсказуемый трафик. Рассмотрите возможность использования потоковой платформы событий (такой, как Kafka).
- База данных должна архивировать все элементы, но предоставляет пользователю только до 1000 элементов. Это наводит на мысль о том, что для архивации всех элементов можно использовать одну базу данных, а для предоставления запрашиваемых элементов — другую. Можно выбрать технологию баз данных, которая лучше всего подходит для каждого сценария использования. Новостной элемент содержит 10 тысяч символов, что составляет 10 Кбайт. В кодировке UTF-8 размер текста составит 40 Кбайт:
  - Для 1000 элементов и 100 тегов общий размер всех новостных элементов составит 1 Гбайт. Такой объем данных легко помещается в кэше Redis.
  - Для архивации можно воспользоваться распределенной шардированной файловой системой, например HDFS.
- Если консистентность в конечном счете с задержкой до нескольких часов приемлема, возможно, устройству пользователя не понадобится обновлять новостные элементы чаще одного раза в час. Таким образом, нагрузка на сервис ленты новостей сокращается.

На рис. 16.2 представлена высокоуровневая архитектура. Очередь и база данных HDFS относятся к категории CDC (отслеживание измененных данных, см. раздел 5.3), тогда как задание ETL, которое читает из HDFS и записывает в Redis, является примером CQRS (разделение ответственности на команды и запросы, см. раздел 1.4.6).

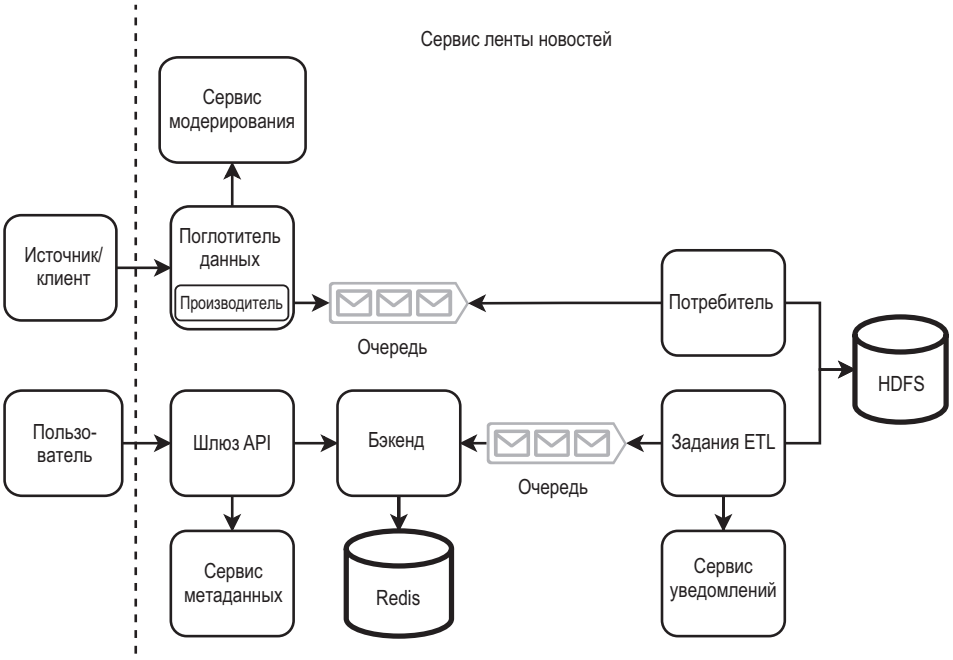

Рис. 16.2. Высокоуровневая архитектура сервиса ленты новостей. Клиент отправляет пост сервису ленты новостей. Сервис поглощения данных получает пост и выполняет простые проверки. Если проверка проходит, сервис поглощения данных отправляет пост в очередь Kafka. Кластер-потребитель потребляет пост и записывает его в HDFS. Пакетные задания ETL обрабатывают посты и отправляют их в другую очередь Kafka. Они также могут инициировать уведомления через сервис уведомлений. Пользовательские запросы на получение постов проходят через шлюз API, который получает теги пользователя от сервиса метаданных, а затем — посты из таблицы Redis через сервис бэкенда. Когда хосты бэкенда бездействуют, они потребляют события из очереди и обновляют таблицу Redis

Источники сервиса ленты новостей передают новые посты поглотителю данных. Поглотитель выполняет задачи проверки в отношении только одного новостного элемента; он не выполняет проверки, зависящие от других новостных элементов или других данных вообще. Примеры таких задач проверки:

- Очистка значений для предотвращения SQL-инъекций.
- Задачи фильтрации и цензуры (например, обнаружение недопустимых выражений). Могут использоваться два набора критериев: элементы, удовлетворяющие критериям из одного набора, немедленно отклоняются, а элементы, удовлетворяющие критериям из другого, — помечаются для ручной проверки. Флаг ручной проверки может присоединяться к элементу перед

его отправкой в очередь. Эта тема более подробно рассматривается в следующем разделе.

- Пост не получен от заблокированного источника/пользователя. Поглотитель получает список заблокированных пользователей из сервиса модерирования. Заблокированные пользователи добавляются в сервис модерирования либо вручную оператором, либо автоматически после определенных событий.
- Обязательные поля имеют ненулевую длину.
- Поле, имеющее максимальную длину, не содержит значение, превышающее эту длину.
- В значении поля, которое не может содержать некоторые символы (например, знаки препинания), отсутствуют такие символы.

Задачи проверки также могут выполняться приложением в клиенте источника перед тем, как он отправит пост поглотителю данных. Однако если какие-то из задач проверки будут пропущены из-за ошибок или вредоносной активности в клиенте, поглотитель может повторить эти проверки. Если какая-либо проверка в поглотителе не проходит, необходимо отправить оповещение разработчикам, чтобы те разобрались, почему клиент и поглотитель возвращают разные результаты проверки.

Запросы некоторых источников должны пройти через сервис аутентификации и авторизации перед тем, как достигнуть поглотителя. На рис. 16.2 они не показаны. Аутентификация OAuth и авторизация OpenID рассматриваются в приложении Б.

Для обработки этого непредсказуемого трафика используется очередь Kafka. Если проверки поглотителя проходят, поглотитель отправляет пост в очередь Kafka и возвращает источнику код успеха 200. Если какая-либо проверка завершится неудачей, поглотитель возвращает источнику код 400 («Неправильный запрос»), а также может включить дополнительную информацию о проверках, которые не прошли.

Потребитель просто выводит элементы из очереди и записывает в HDFS. Понадобится минимум две таблицы HDFS: для необработанных элементов, отправленных потребителем, и для новостных элементов, готовых к выводу для пользователей. Также понадобится отдельная таблица для элементов, которые должны пройти ручную проверку, прежде чем выводиться пользователям. Скорее всего, система ручной проверки подробно обсуждаться на собеседовании не будет. Эти таблицы HDFS секционируются по тегам и часам.

Пользователи отправляют запросы GET/post к шлюзу API, который обращается к сервису метаданных за тегами пользователя, а затем запрашивает соответствующие новостные элементы из кэша Redis через сервис бэкенда. Ключом кэша Redis может быть кортеж `(tag, hour)` (тег, час), а значением — соответствующий

список новостных элементов. Эту структуру данных можно представить в формате `{(tag, hour), [post]}`, где `tag` — строка, `hour` — целое число, а `post` — объект, содержащий строку идентификатора поста и строку тела/контента.

Шлюз API также выполняет стандартные обязанности, описанные в разделе 6.1, включая аутентификацию и авторизацию, а также ограничение частоты запросов. Если количество хостов возрастает до большого значения, а стандартные обязанности фронтенда и запросы к сервису метаданных и сервису Redis используют разные аппаратные ресурсы, функциональности запросов можно выделить в отдельный сервис бэкенда, чтобы масштабировать их независимо.

Что касается требования к консистентности в конечном счете и нашего наблюдения о том, что устройству пользователя может не требоваться обновлять новостные элементы более одного раза в час, если пользователь запрашивает обновление в пределах часа от своего предыдущего запроса, нагрузку на сервис можно сократить одним из двух способов:

1. Устройство может проигнорировать запрос.
2. Устройство может отправить запрос, но не делать повторную попытку при получении ответа 504 («Тайм-аут»).

Задания ETL записывают в другую очередь Kafka. Когда хосты бэкенда не обслуживают пользовательские запросы на получение постов, они могут потреблять из очереди Kafka и обновлять таблицу Redis. Задания ETL выполняют функции, описанные ниже.

Прежде чем выводить для пользователей необработанные новостные элементы, возможно, сначала придется выполнить проверку данных или операции модерирования/цензуры, зависящие от других новостных элементов или других элементов вообще. Для простоты будем обозначать эти задачи общим термином «задачи проверки». На рис. 16.3 они могут быть параллельными задачами ETL. Для каждой задачи может понадобиться дополнительная таблица HDFS. В каждой таблице хранятся идентификаторы элементов, прошедших проверку. Примеры:

- Поиск дубликатов элементов.
- Если количество новостных элементов для конкретного тега/темы, которое можно отправить за час, ограниченно, для этого ограничения также может существовать задача проверки.
- Определение пересечения идентификаторов элементов из промежуточных таблиц HDFS, то есть набора идентификаторов, прошедших все проверки. Полученный набор записывается в итоговую таблицу HDFS. Идентификаторы читаются из итоговой таблицы HDFS, после чего соответствующие новостные элементы копируются для перезаписи кэша Redis.

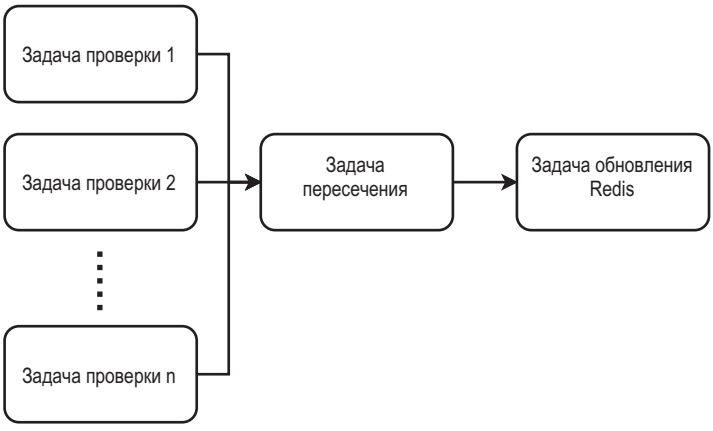

Рис. 16.3. Направленный ациклический граф задания ETL. Задачи проверки выполняются параллельно. Каждая задача выводит набор идентификаторов проверенных постов. Когда задачи выполнены, задача пересечения определяет пересечение всех этих наборов, то есть идентификаторы постов, которые можно выводить для пользователей

Можно создать задания ETL для инициирования уведомлений через сервис уведомлений. Каналы уведомлений могут включать мобильные и браузерные приложения, электронную почту, текстовые сообщения и социальные сети. Подробное обсуждение сервиса уведомлений см. в главе 9. В этой главе он подробно рассматриваться не будет.

Обратите внимание на ключевую роль модерирования в сервисе ленты новостей, как показано на рис. 16.2 и обсуждается в контексте заданий ETL. Возможно, также потребуется модерирование постов каждого конкретного пользователя, например, как уже говорилось, заблокированным пользователям должно быть запрещено отправлять запросы. На рис. 16.4 показано объединение всех задач модерирования в один сервис модерирования. Эта тема более подробно обсуждается в разделе 16.4.

16.3. ПРЕДВАРИТЕЛЬНАЯ ПОДГОТОВКА ЛЕНТЫ

На рис. 16.2 каждому пользователю понадобится один запрос Redis на пару `(tag, hour)`. Каждому пользователю для получения интересующих его элементов может понадобиться много запросов, что приведет к высокому трафику чтения, и возможно, высокой задержке сервиса ленты новостей.

Можно выбрать решение с увеличенными затратами памяти для снижения задержки и трафика и подготовить ленту пользователя заранее. Можно подготовить две хеш-карты, `{user ID, post ID}` и `{post ID, post}`. Если предположить, что в системе поддерживается 100 тегов с 1000 элементов в 10 000 символов каждый,

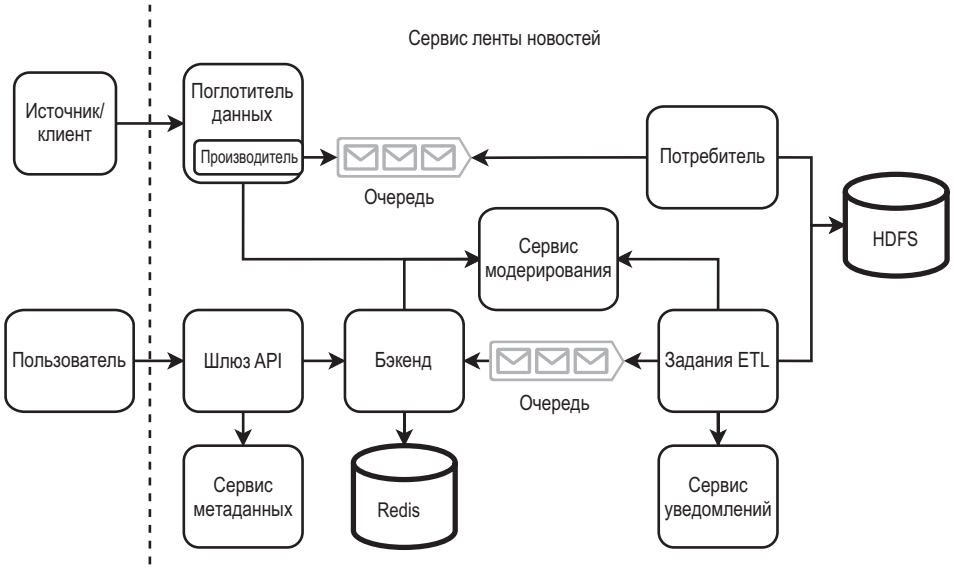

Рис. 16.4. Высокоуровневая архитектура системы ленты новостей с централизацией модерирования всего контента в сервисе модерирования. Разработчики могут определять всю логику модерирования в этом сервисе

последняя хеш-карта займет чуть более 1 Гбайт. Для первой хеш-карты необходимо хранить 1 миллиард идентификаторов пользователей и до 100 × 1000 возможных идентификаторов постов. Идентификатор занимает 64 бита. Общие требования к объему хранилища составляют до 800 Тбайт, что может превысить емкость кластера Redis. Одно из возможных решений — секционировать пользователей по регионам и хранить всего два-три региона на датацентр; таким образом, один датацентр будет обслуживать до 20 миллионов пользователей, что составляет 16 Тбайт. Другое возможное решение — ограничить необходимый объем хранилища до 1 Тбайт, введя лимит до нескольких десятков идентификаторов постов, но это нарушает требование о 1000 элементов.

Еще одно возможное решение — использование шардированной реализации SQL для пары {идентификатор пользователя, идентификатор поста}, как обсуждалось в разделе 4.3. Таблицу можно шардировать по хешированным идентификаторам пользователей, чтобы идентификаторы случайным образом распределялись между узлами, а следовательно, наиболее активные пользователи тоже распределялись случайным образом. Это позволит избежать проблемы «горячих шардов». Когда бэкенд получает запрос на извлечение постов для идентификатора пользователя, он хеширует идентификатор пользователя, а затем отправляет запрос к нужному узлу SQL. (Вскоре мы обсудим, как он находит нужный узел SQL.) Таблица, содержащая пары `{post ID, post}`, может реплицироваться на

всех узлах, так что между этими двумя таблицами можно выполнять запросы JOIN. (Таблица также может содержать другие столбцы измерений для меток времени, тегов и т. д.) На рис. 16.5 представлена наша стратегия шардирования и репликации.

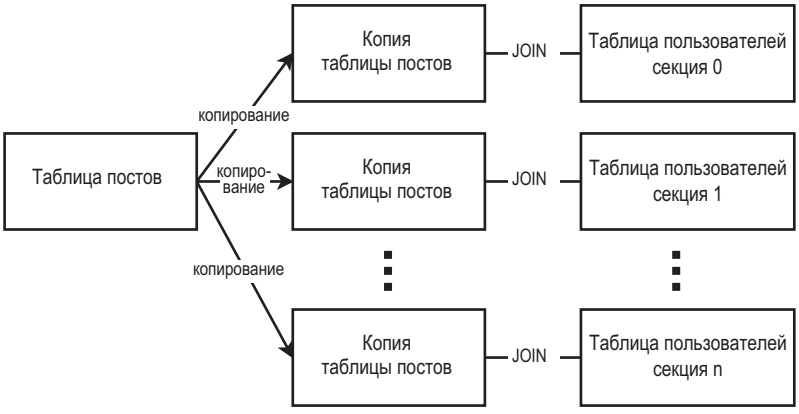

Рис. 16.5. Используемая стратегия шардирования и репликации. Таблица с парами {hashed user ID, post ID} шардируется и распределяется по нескольким хостам-лидерам и реплицируется на хосты-последователи. Таблица с {post ID, post} реплицируется на всех хостах. Можно выполнять операцию JOIN по идентификатору поста

Как показано на рисунке (рис. 16.6), 64-разрядное адресное пространство хешированных идентификаторов пользователей распределяется по кластерам. Кластер 0 содержит любые хешированные идентификаторы пользователей в диапазоне $[0, (2^{64} - 1)/4)$, кластер 1 — любые хешированные идентификаторы пользователей в диапазоне $[(2^{64} - 1)/4, (2^{64} - 1)/2)$, кластер 2 — любые хешированные идентификаторы пользователей в диапазоне $[(2^{64} - 1)/2, 3 \times (2^{64} - 1)/4)$ и кластер 3 — любые хешированные идентификаторы пользователей в диапазоне $[3 \times (2^{64} - 1)/4, 2^{64} - 1)$. Начать можно с такого равномерного разбиения. Когда между кластерами возникнут разные уровни трафика, трафик можно сбалансировать, регулируя количество и размеры секций.

Как бэкенд найдет подходящий узел SQL? Необходимо отображать хешированные идентификаторы пользователей на имена кластеров. Каждый кластер должен содержать несколько адресных записей, по одной для каждого последователя, так что хост бэкенда случайным образом назначается узлу-последователю в соответствующем кластере.

Необходимо отслеживать уровень трафика к кластерам, чтобы выявить горячие шарды и перебалансировать трафик соответствующим изменением размеров кластера. Можно отрегулировать емкость жесткого диска хоста для экономии затрат. Если вы используете облачный провайдер, можно отрегулировать размер используемой виртуальной машины.

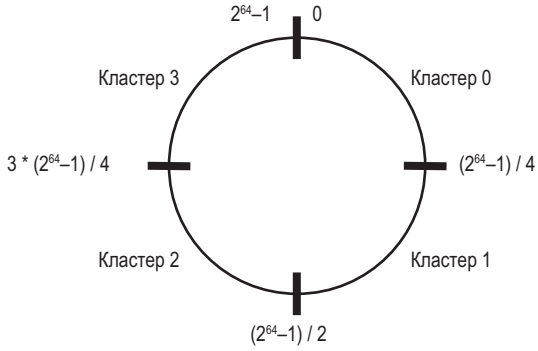

Рис. 16.6. Согласованное хеширование хешированных идентификаторов пользователей по именам кластеров. Кластеры можно разделить по 64-разрядному пространству адресов. На этой схеме показаны четыре кластера, каждый из которых занимает четверть пространства адресов. Можно начать с равного разбиения, а затем отрегулировать количество и размеры секций, чтобы сбалансировать трафик между ними

На рис. 16.7 изображена высокоуровневая архитектура сервиса ленты новостей с этой структурой. Когда пользователь отправляет запрос, бэкенд хеширует идентификатор пользователя, как было описано выше. Затем бэкенд обращается к ZooKeeper, чтобы получить имя кластера, и отправляет кластеру запрос SQL. Запрос отправляется случайном узлу-последователю, выполняется на нем, после чего пользователю возвращается результат — список постов.

Если единственным клиентом является мобильное приложение (то есть веб-приложение отсутствует), можно сэкономить память за счет хранения постов на стороне клиента. Тогда можно исходить из того, что пользователю будет достаточно загрузить свои посты всего один раз, и удалить строки после их получения. Если пользователь входит с другого мобильного устройства, он не увидит посты, которые были загружены на предыдущем устройстве. Это достаточно редкий сценарий, поэтому сочтем его приемлемым, особенно если учесть, что новости быстро устаревают и через несколько дней после публикации уже не представляют особого интереса для пользователей.

Также можно добавить столбец с меткой времени и создать задание ETL, которое периодически удаляет строки старше 24 часов.

Наконец, можно обойтись без шардирования SQL за счет объединения обоих подходов. Когда пользователь открывает мобильное приложение, можно воспользоваться заранее подготовленной лентой, чтобы обслужить только первый запрос его постов, и хранить только такое количество идентификаторов постов, которое помещается на одном узле. Когда пользователь прокручивает список, приложение может отправлять запросы на получение дополнительных постов; эти запросы могут обслуживаться из Redis. На рис. 16.8 представлена высокоуровневая архитектура такого решения с Redis. Это

способ с повышением сложности и затрат на обслуживание в целях снижения задержки и затрат.

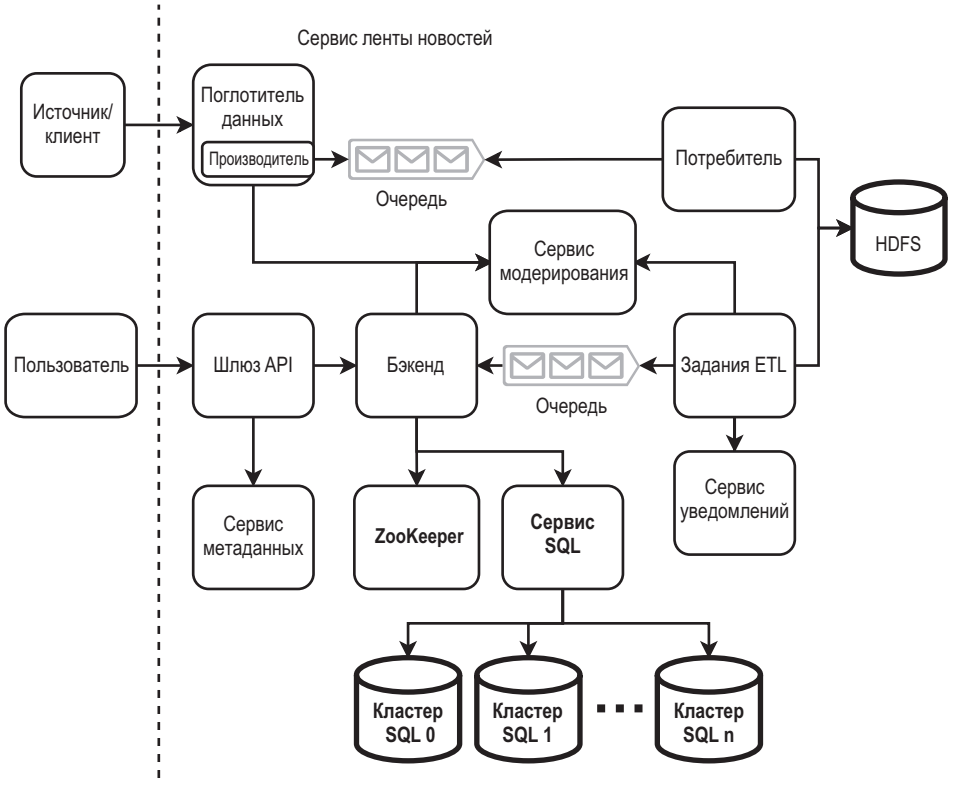

Рис. 16.7. Высокоуровневая архитектура системы ленты новостей с заранее подготовленной лентой. Отличия от рис. 16.4 выделены жирным шрифтом. При получении пользовательского запроса бэкенд сначала получает соответствующее имя кластера SQL, после чего запрашивает посты у соответствующего кластера SQL. Бэкенд может направлять пользовательские запросы узлу-последователю, содержащему запрашиваемый идентификатор пользователя. Кроме того, как показано на схеме, маршрутизацию запросов SQL можно выделить в сервис SQL

Обсудим два способа избежать многократной загрузки клиентом одних и тех же постов из Redis:

1. Клиент может включить идентификаторы поста, которые содержатся в его запросе GET/post, чтобы бэкенд вернул посты, еще не загруженные клиентом.
2. Таблица Redis помечает посты по часам. Клиент может запросить посты за определенный час. Если возвращается слишком много сообщений, приращe-

ния времени можно уменьшить (например, до 10-минутных блоков в час). Другой возможный способ — предоставить конечную точку API, которая возвращает все идентификаторы постов за определенный час, и тело запроса конечной точки GET/post, в котором пользователь может указать идентификаторы постов, которые он желает загрузить.

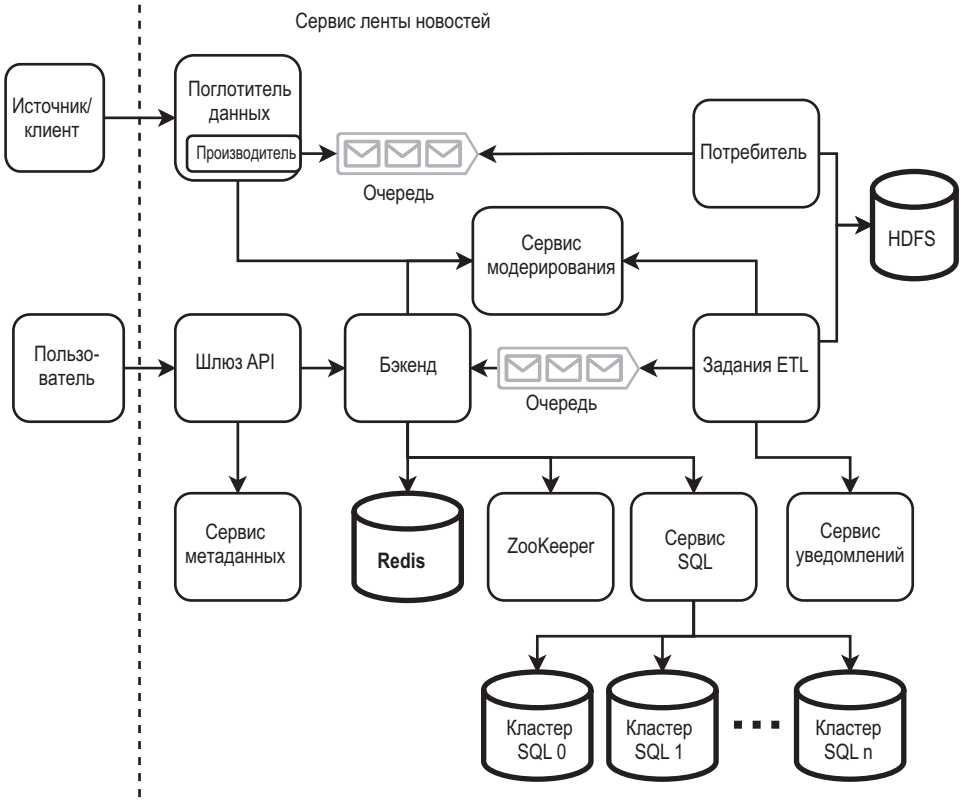

Рис. 16.8. Высокоуровневая архитектура с предварительной подготовкой ленты новостей и сервисом Redis. Отличается от рис. 16.7 добавлением сервиса Redis (выделен жирным шрифтом)

16.4. ПРОВЕРКА ДАННЫХ И МОДЕРИРОВАНИЕ КОНТЕНТА

В этом разделе рассматриваются проблемы, связанные с проверкой данных (валидацией), и их возможные решения. Проверка может не обнаружить всех проблем, и некоторые посты будут ошибочно доставлены пользователям. Правила фильтрации контента могут различаться в зависимости от демографии пользователей.

В разделе 15.6 был описан сервис подтверждений для Airbnb, представляющий еще один подход к проверке данных и модерированию контента. Здесь можно кратко обсудить эту тему. На рис. 16.9 изображена высокоуровневая архитектура с сервисом подтверждений. Задания ETL помечают некоторые посты для ручной проверки. Такие посты отправляются сервису подтверждения для ручной проверки. Если ревьюер одобряет пост, он будет отправлен в очередь Kafka, потреблен бэкендом и выведен для пользователей. Если ревьюер отклоняет пост, то сервис подтверждений уведомляет об этом источник/клиент через сервис обмена сообщениями.

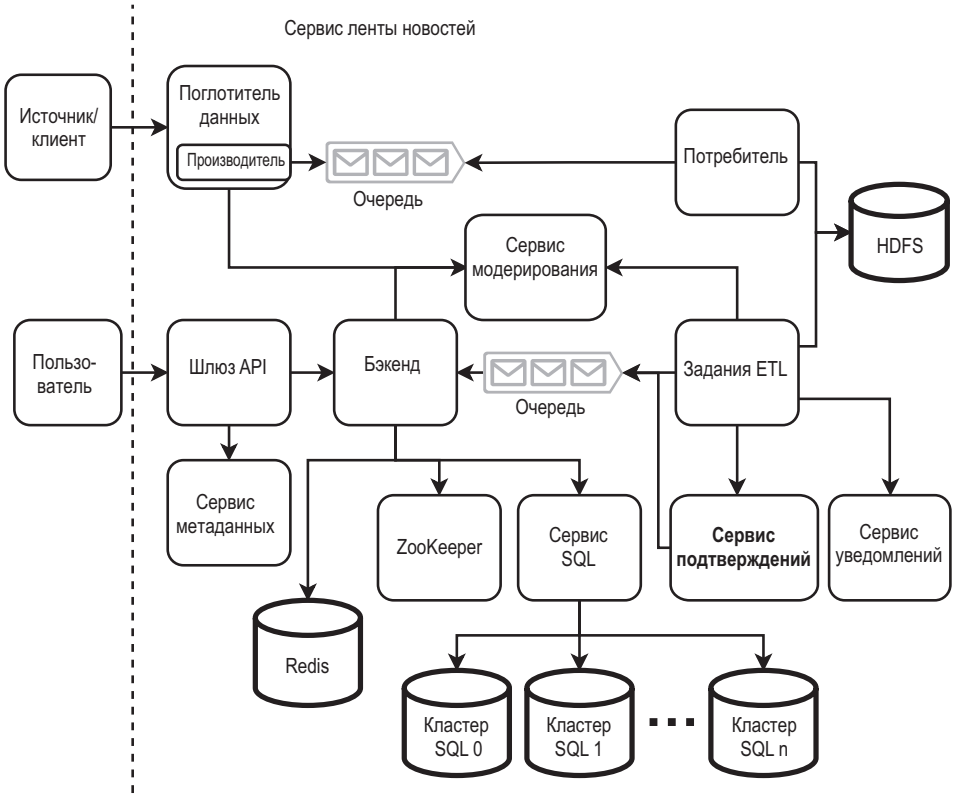

Рис. 16.9. Задания ETL помечают некоторые посты для ручной проверки. Эти посты отправляются сервису подтверждения (выделен жирным шрифтом; этот сервис добавляется к рис. 16.8) для ручной проверки вместо отправки в очередь Kafka. (Если частота постов, помеченных для проверки, высока, возможно, сам сервис подтверждений должен содержать очередь Kafka.) Если ревьюер одобрит пост, он отправляется обратно в очередь Kafka через задание ETL для потребления бэкендом (также ревьюер может отправить пост на бэкенд напрямую). Если ревьюер отклоняет пост, сервис подтверждений уведомляет источник/клиент через сервис обмена сообщениями (на диаграмме не показан)

16.4.1. Изменение постов на устройствах пользователей

Некоторые проверки трудно автоматизировать. Например, пост может быть усечен. Для простоты рассмотрим пост, состоящий из одного предложения: This is a post. Усеченный пост может иметь вид: This is a[1]. Пост с орфографическими ошибками легко обнаружить, однако этот пост не содержит орфографических ошибок, но при этом он очевидно недействителен. Такие проблемы сложно выявить автоматической проверкой.

Некоторые виды недопустимого контента (например, запрещенные слова) легко обнаружить, но большую часть такого контента (например, неподходящий по возрасту, угрозы минирования или фейковые новости) выявить автоматическими средствами чрезвычайно сложно.

В любом дизайне системы не стоит пытаться избежать всех ошибок и сбоев. Следует исходить из того, что ошибки и сбои неизбежны, и разрабатывать механизмы, которые упрощают их обнаружение, диагностику и исправление. Некоторые посты, которые не должны быть доставлены пользователям, могут быть доставлены по ошибке. Понадобится механизм для удаления таких постов из сервиса ленты новостей или перезаписи их исправленными постами. Если устройства пользователей кэшируют посты, они должны быть удалены из кэша или перезаписаны исправленными версиями.

Для этого можно изменить конечную точку `GET /posts`. Каждый раз, когда пользователь получает посты, ответ должен содержать список исправленных постов и список постов на удаление. Клиентское мобильное приложение должно вывести исправленные посты и удалить те, которые должны быть удалены.

Один из возможных способов основан на добавлении в пост перечисления event с возможными значениями `REPLACE` и `DELETE`. Если вы хотите заменить или удалить старый пост на стороне клиента, следует создать новый объект поста с таким же идентификатором, как у старого. Объект поста должен иметь событие со значением `REPLACE` для замены или `DELETE` для удаления.

Чтобы сервис ленты новостей понимал, какие посты на стороне клиента должны быть изменены, он должен знать, какие посты хранятся в клиенте. Сервис ленты новостей может вести журнал идентификаторов постов, загруженных клиентом, но требования к хранению могут оказаться слишком масштабными и затратными. Если установить период удержания данных клиентами (допустим, 24 часа или 7 дней), чтобы старые посты автоматически удалялись, старые журналы тоже можно удалять, но хранение данных все равно может обойтись дорого.

---

[1] This is a post — «Это — пост». This is a можно прочесть как «Это — а». — *Примеч. пер.*

Еще одно возможное решение — включение клиентами идентификаторов текущих постов в запросы GET/post. Бэкенд обрабатывает эти идентификаторы постов и определяет, какие новые посты следует отправить (как обсуждалось выше), а какие изменить или удалить.

В разделе 16.4.3 обсуждается сервис модерирования, одна из ключевых функций которого — просмотр текущих доступных постов администраторами и принятие решений об изменении или удалении постов.

16.4.2. Назначение тегов постам

Будем исходить из предположения, что подтверждается или отклоняется пост целиком. Иначе говоря, если какая-то часть поста не проходит проверку или модерирование, отклоняется весь пост, и часть, прошедшая проверку, также не выводится. Что делать с постами, не прошедшими проверку? Можно просто удалить их, уведомить источники или провести проверку вручную. Первый вариант может ухудшить пользовательский опыт, а третий в большом масштабе может оказаться слишком затратным. Мы выберем второй вариант.

Расширим задачу пересечения на рис. 16.3, чтобы она также сообщала ответственному источнику/пользователю, если какая-то проверка не проходит. Задача пересечения может объединить все неудачные проверки и отправить их источнику/пользователю в одном сообщении. Она может использовать для отправки общий сервис обмена сообщениями. Каждая задача проверки может иметь идентификатор и короткое описание проверки. Сообщение может содержать идентификаторы и описания неудачных проверок на случай, если пользователь захочет связаться с вашей компанией, чтобы уточнить, какие изменения надо внести в пост, или оспорить решение о блокировке.

Другое требование, которое стоит обсудить, — нужно ли различать глобально и регионально применимые правила. Некоторые правила могут действовать только в конкретных странах из-за местных культурных особенностей или особенностей нормативно-правового регулирования. Говоря в общем, не следует выводить для пользователя определенные посты в зависимости от заданных им предпочтений и демографических показателей (таких, как возраст или регион). Кроме того, такие посты нельзя отклонять в поглотителе данных, поскольку в этом случае задачи проверки будут применяться ко всем, а не только к отдельным пользователям. Вместо этого необходимо назначить постам специальные теги — метаданные, которые будут использоваться для фильтрации определенных постов для каждого пользователя. Чтобы избежать путаницы с тегами пользовательских интересов, будем называть такие теги фильтрующими тегами, или, сокращенно, фильтрами. Таким образом, пост может иметь как теги, так и фильтры. Ключевое различие между тегами и фильтрами состоит в том, что пользователи сами настраивают теги предпочтений, тогда как фильтры полностью

контролируют разработчики. Как показано в следующем разделе, это означает, что фильтры будут настраиваться в сервисе модерирования, а теги — нет.

Будем считать, что при добавлении нового тега/фильтра или удалении текущего тега/фильтра изменение будет распространяться только на будущие посты и изменения разметки прошлых постов не потребуется.

Одного обращения к Redis недостаточно для получения всех постов пользователя. Понадобятся три хеш-таблицы Redis со следующими парами «ключ — значение»:

- `{post ID, post}`: для получения постов по идентификатору.
- `{tag, [post ID]}`: для отбора идентификаторов постов по тегу.
- `{post ID, [filter]}:` для исключения постов по фильтру.

Потребуется несколько операций выборки «ключ — значение». Последовательность действий выглядит так:

1. Клиент отправляет запрос GET/post сервису ленты новостей.
2. Шлюз API запрашивает у сервиса метаданных теги и фильтры клиента. Клиент также может хранить свои теги и фильтры и предоставлять их по запросу GET/post; тогда эту операцию выборки можно пропустить.
3. Шлюз API запрашивает у Redis идентификаторы постов с тегами и фильтрами пользователей.
4. Шлюз API запрашивает у Redis фильтр для каждого идентификатора поста и исключает этот идентификатор поста из результатов пользователя, если пост содержит какие-либо из фильтров пользователя.
5. Шлюз API запрашивает у Redis пост для каждого идентификатора поста, а затем возвращает полученные посты клиенту.

Обратите внимание: *логика исключения идентификаторов постов по тегам должна выполняться на уровне приложения*. Альтернатива — использование таблиц SQL вместо таблиц Redis. Можно создать таблицу постов со столбцами (post_id, post), таблицу тегов со столбцами (tag, post_id) и таблицу фильтров со столбцами (filter, post_id), после чего выполнить один запрос SQL JOIN для получения постов клиента:

```
SELECT post
FROM post p JOIN tag t ON p.post_id = t.post_id
LEFT JOIN filter f ON p.post_id = f.post_id
WHERE p.post_id IS NULL
```

В разделе 16.3 обсуждалась предварительная подготовка ленты пользователя с таблицей Redis {user_id, post_id}. Даже с учетом требований фильтрации

постов, рассмотренными в этом разделе, можно создать задание ETL, которое подготавливает эту таблицу Redis.

Наконец, для лент новостей с региональной привязкой может возникнуть необходимость секционировать кэш Redis по региону или включить в ключ Redis дополнительный столбец region. Так же можно поступить, если нужно поддерживать несколько языков.

16.4.3. Сервис модерирования

Наша система выполняет проверку данных в четырех местах: в клиенте, поглотителе данных, заданиях ETL и бэкенде во время запросов GET/post. Те же проверки будут реализованы в разных браузерных и мобильных приложениях и в поглотителе, хотя это и означает дублирование усилий по разработке и обслуживанию и повышение риска ошибок. Проверки повышают нагрузку на процессор, но сокращают трафик к сервису ленты новостей, что означает уменьшение размера кластера и снижение затрат. Кроме того, это решение более безопасно. Если взломщики обойдут проверки на стороне клиента, отправляя запросы API напрямую к сервису ленты новостей, проверки на стороне сервиса перехватят недействительные запросы.

Что касается проверок на стороне сервера, поглотитель, задания ETL и бэкенд используют разные проверки. Но как показано на рис. 16.4, можно консолидировать и абстрагировать их в один сервис — сервис модерирования.

Как было сказано в предыдущем подразделе, посвященном тегам и фильтрам, общая цель сервиса модерирования — сделать так, чтобы разработчики (не пользователи) управляли тем, увидят ли пользователи отправленные посты. Исходя из сказанного выше, сервис модерирования предоставляет администраторам следующую функциональность:

1. Настройка задач проверки данных (валидации) и фильтров.
2. Выполнение решений об изменении или удалении постов.

Объединение задач модерирования в один сервис гарантирует, что команды, работающие над разными сервисами внутри сервиса ленты новостей, случайно не реализуют одинаковые проверки, и позволит нетехническим специалистам из групп модерирования контента выполнять все задачи модерирования без помощи инженеров. Сервис модерирования также регистрирует эти решения в целях проверки, аудита и отката (отмены решений модерирования).

Запрос модерирования может обрабатываться так же, как другие запросы записи в сервис ленты новостей. По аналогии с заданиями ETL, сервис модерирования отправляет событие в тему ленты новостей, а сервис ленты новостей потребляет это событие и записывает соответствующие данные в Redis.

Инструменты коммуникаций

В общем случае взаимодействие с командами инженеров и приоритетное выполнение инженерных задач организовать сложно, особенно в больших компаниях. Любые инструменты, позволяющие выполнять работу без такого взаимодействия, обычно стоят вложенных средств.

16.5. ВЕДЕНИЕ ЖУРНАЛОВ, МОНИТОРИНГ И ОПОВЕЩЕНИЯ

В разделе 2.5 обсуждались ключевые концепции ведения журналов, мониторинга и оповещения, которые необходимо упомянуть на собеседовании. В дополнение к сказанному в разделе 2.5 отслеживать и отправлять оповещения необходимо в следующих случаях.

- Аномально высокий или низкий трафик от любого конкретного источника.
- Аномально высокая доля элементов, не проходящих проверку данных, — как в целом, так и для каждого отдельного источника.
- Отрицательные реакции пользователей, например пометка статей как ошибочных или оскорбительных.
- Аномально долгая обработка элементов в пайплайне. Можно отслеживать сравнением метки времени отправки элемента с временем, в которое элемент попадает в базу Redis. Аномально долгая обработка может указывать на необходимость масштабирования некоторых компонентов пайплайна или на неэффективность операций пайплайна, и это стоит проанализировать.

16.5.1. Поддержка изображений

В следующей версии новостные элементы могут содержать 0–10 изображений размером до 1 Мбайт каждое. Изображения будут рассматриваться как часть объекта поста, а тег или фильтр применяется ко всему объекту поста, а не только к отдельным его свойствам (таким, как тело поста или отдельное изображение).

Новые требования значительно повышают затраты на отправку запросов GET /post. Файлы изображений существенно отличаются от строк тела поста:

- Файлы изображений намного больше тела поста, и для них требуются другие технологии хранения данных.
- Файлы изображений могут повторно использоваться между постами.
- Скорее всего, алгоритмы проверки данных для графических файлов будут использовать библиотеки обработки изображений, которые существенно отличаются от проверки строк тела поста.

16.5.2. Высокоуровневая архитектура

Для начала заметим, что требование, ограничивающее текст статьи 40 Кбайт, пренебрежимо мало по сравнению с требованием 10 Мбайт для содержащейся в нем графики. Это означает, что загрузка или обработка текста статьи будет выполняться быстро, но загрузка или обработка изображений будет занимать больше времени и вычислительных ресурсов.

На рис. 16.10 представлена высокоуровневая архитектура с сервисом мультимедиа. Загрузка мультимедийных данных должна выполняться синхронно,

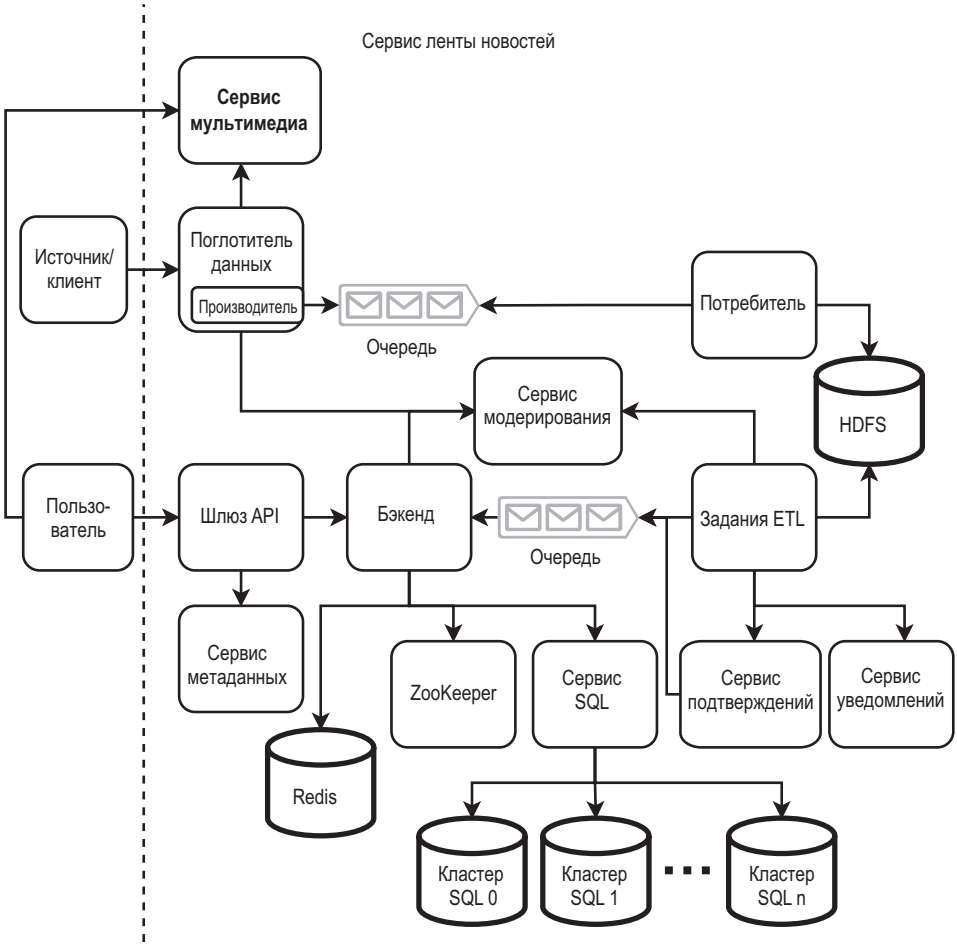

Рис. 16.10. Добавление сервиса мультимедиа (выделен жирным шрифтом и линиями; этот сервис добавлен к рис. 16.9), что позволяет включать в новостные элементы графику, аудио и видео. Выделение сервиса мультимедиа также упрощает управление и анализ данных отдельно от новостных элементов

поскольку источнику необходимо сообщить, прошла ли она успешно или завершилась ошибкой. Это означает, что кластер сервиса поглотителя данных будет намного больше, чем до добавления графики в статьи. Сервис мультимедиа хранит данные в общем сервисе объектов, реплицируемом по нескольким датацентрам, чтобы пользователь мог обращаться к мультимедийным данным в ближайшем к нему датацентре.

На рис. 16.11 представлена диаграмма последовательности действий при загрузке статьи источником. Так как загрузка мультимедиа требует большего объема передачи данных, чем загрузка метаданных или текста, загрузку метаданных необходимо завершить перед отправкой метаданных статей и текста в очередь Kafka. Если загрузка завершится успешно, но отправка в очередь окажется неудачной, можно вернуть источнику ошибку 500. В ходе загрузки файла в сервис мультимедиа поглотитель сначала хеширует файл, а затем отправляет хеш сервису мультимедиа, чтобы тот проверил, был ли файл загружен ранее. Если он уже загружен, сервис мультимедиа возвращает поглотителю ответ 304, и затратная пересылка по сети не проводится. В этом дизайне кластер-потребитель намного меньше кластера сервиса мультимедиа.

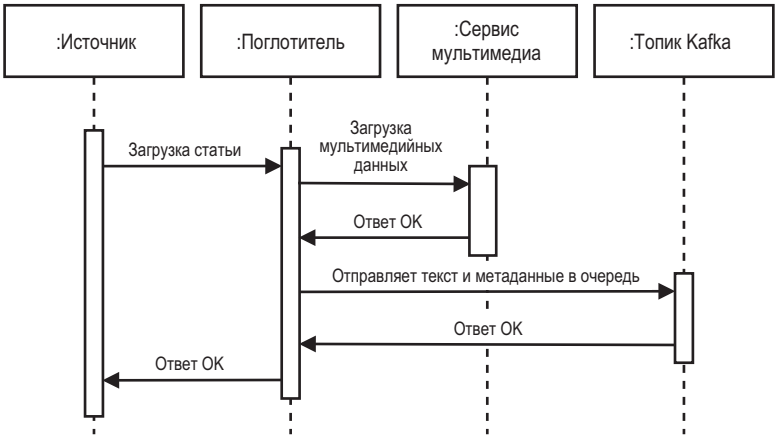

Рис. 16.11. Диаграмма последовательности действий при загрузке статьи. Мультимедийные данные почти всегда занимают больше места, чем текст, поэтому их поглотитель загружает в сервис мультимедиа в первую очередь. После успешной загрузки мультимедийных данных поглотитель отправляет текст и метаданные в топик Kafka для поглощения и записи в HDFS, как описано в этой главе

А что, если на хосте поглотителя данных произойдет сбой уже после успешной загрузки мультимедийных данных, но до отправки в топик Kafka? Целесообразно сохранить загрузку, а не удалять ее, поскольку загрузка мультимедийных данных требует значительных ресурсов. Источник получит ответ с признаком ошибки и может повторить загрузку. На этот раз сервис мультимедиа вернет

код 304, как говорилось в предыдущем разделе, и поглотитель сгенерирует соответствующее событие. Впрочем, источник может не сделать повторной попытки. В этом случае можно периодически выполнять задание аудита для нахождения мультимедийных данных, не имеющих связанных метаданных и текста в HDFS, и удалять эти данные.

Если пользователи широко разбросаны географически или пользовательский трафик слишком интенсивен для вашего сервиса мультимедиа, можно воспользоваться CDN. Проектирование систем CDN рассмотрено в главе 13. Токены авторизации для загрузки изображений из CDN могут предоставляться шлюзом API с использованием архитектуры сервисной сети. На рис. 16.12 представлена высокоуровневая архитектура с CDN. Новый элемент содержит текстовые поля для контента (такие, как заголовок, тело и URL-адреса мультимедийных данных). На рис. 16.12 источник отправляет изображения сервису изображений, а текстовый контент — сервису ленты новостей. Клиент может:

- загрузить текст статьи и URL мультимедийных данных из Redis;
- загрузить мультимедийные данные из CDN.

Основные отличия от рис. 16.10:

- Сервис мультимедиа записывает мультимедийные данные в CDN, а пользователи загружают их из CDN.
- Задания ETL и сервис подтверждения отправляют запросы к сервису мультимедиа.

Мы используем как сервис мультимедиа, так и CDN, поскольку часть статей не будет выводиться для пользователей, поэтому некоторые изображения не обязательно хранить в CDN, и это позволит сократить затраты. Часть заданий ETL могут служить автоматическими подтверждениями статей, так что эти задания должны информировать сервис мультимедиа, что статья прошла проверку, и сервис мультимедиа должен отправить мультимедийные данные статьи в CDN, чтобы вывести их для пользователей. Сервис подтверждения отправляет похожие запросы сервису мультимедиа.

Можно обсудить плюсы и минусы хранения и работы с текстом и мультимедиа в разных сервисах по сравнению с одним сервисом. В главе 13 более подробно обсуждается размещение изображений в CDN, например достоинства и недостатки размещения мультимедиа.

На следующем шаге также можно хранить в CDN статьи целиком, включая весь текст и мультимедиа. Значения Redis могут быть сокращены до идентификаторов статей. Хотя текст статьи обычно намного меньше ее мультимедийных данных, ее размещение в CDN все еще может обеспечить повышение эффективности, особенно для частых запросов популярных статей. Redis масштабируется горизонтально, но репликация между датацентрами становится сложной задачей.

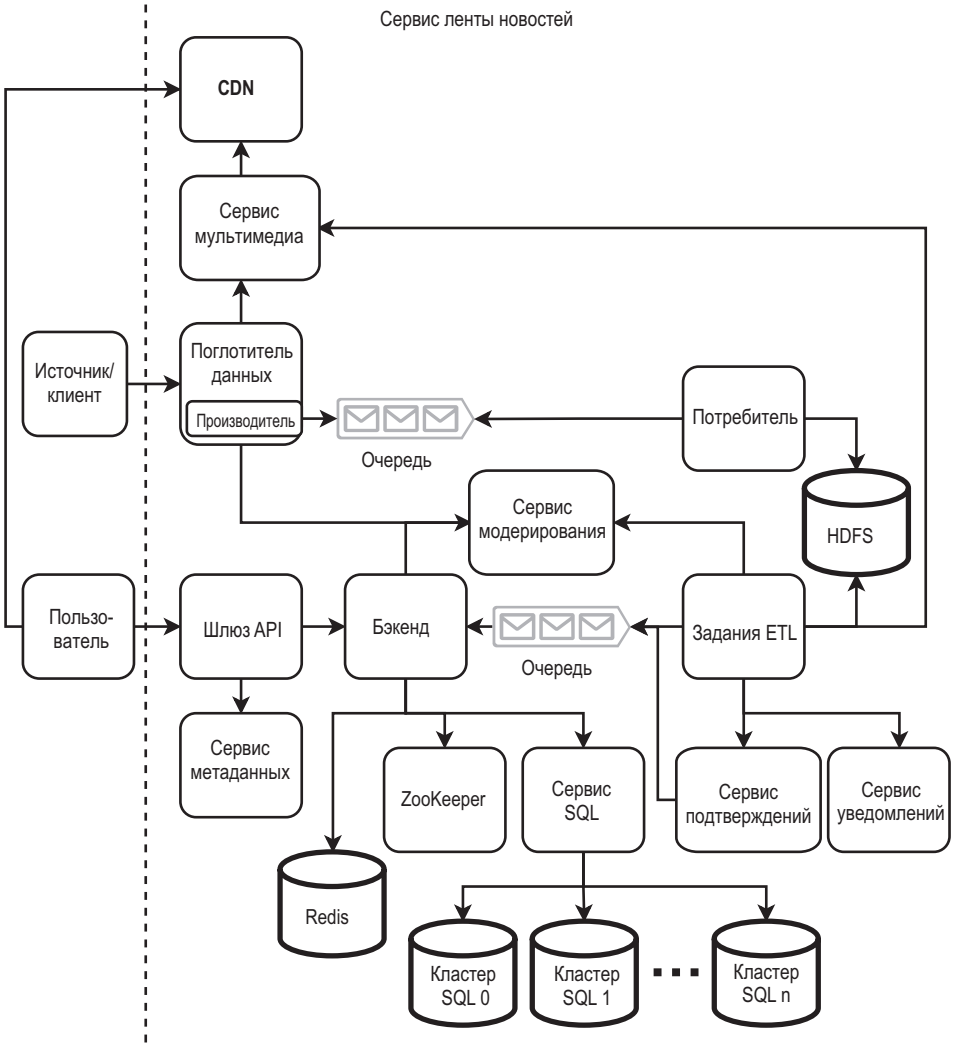

Рис. 16.12. Использование CDN (компонент выделен жирным; этот сервис добавляется к рис. 16.10) для размещения мультимедийных данных. Пользователи загружают изображения прямо из CDN, пользуясь такими преимуществами CDN, как низкая задержка и более высокая доступность

Как проверять изображения и текст статьи в сервисе подтверждения — по отдельности или вместе? Для простоты будем считать, что проверка статьи состоит из единой проверки текста и мультимедийных данных.

Как повысить эффективность проверки мультимедиа? Нанимать ревьюеров дорого, и такому человеку придется полностью прослушать аудио или просмотреть

видео, прежде чем принять решение. Можно перевести аудио в текст, чтобы ревьюер мог прочитать его, вместо того чтобы слушать. Это позволит привлекать людей с дефектами слуха, повысив культуру инклюзивности в компании. В ходе проверки сотрудник может воспроизводить видео на скорости 2x или 3x и читать текст отдельно от просмотра видеофайла. Также для проверки статей можно применять машинное обучение.

16.6. ДРУГИЕ ВОЗМОЖНЫЕ ТЕМЫ ОБСУЖДЕНИЯ

Еще несколько тем, которые эксперт или кандидат может предложить в ходе собеседования:

- создание динамических хештегов вместо фиксированного набора тем;
- отправка новостных элементов другим пользователям или группам;
- более подробное обсуждение отправки уведомлений авторам и читателям;
- публикация статей в реальном времени. Задания ETL должны быть потоковыми, а не пакетными;
- повышение приоритета одних статей относительно других.

Можно рассмотреть вопросы, которые не вошли в обсуждение функциональных требований:

- аналитика;
- персонализация. Вместо предоставления всем пользователям 1000 одинаковых новостных элементов вы предоставляете каждому пользователю персонализированный набор из 100 элементов. Такой дизайн будет существенно более сложным;
- публикация статей на других языках, кроме английского. Потенциальные сложности (например, работа с UTF или языковые транзакции);
- монетизация ленты новостей. Возможные темы:
  - проектирование системы подписки;
  - резервирование определенных постов для подписчиков;
  - ограничение по количеству статей для пользователей, которые не являются подписчиками;
  - реклама и продвигаемые посты.

ИТОГИ

- При составлении схемы исходной высокоуровневой архитектуры для системы ленты новостей учитывайте важнейшие требуемые данные и включите в схему компоненты, которые читают и записывают эти данные в базу.

- Рассмотрите нефункциональные требования к чтению и записи данных. Выберите подходящие типы баз данных и вспомогательные сервисы, если они есть. К их числу относятся сервисы Kafka и Redis.
- Определите, какие операции не требуют низкой задержки, и поместите их в пакетные и потоковые задания, чтобы обеспечить масштабируемость.
- Определите операции обработки, которые должны выполняться до и после чтения и записи. Оберните их в сервисы. К операциям, выполняемым до, могут относиться сжатие, модерирование контента и обращения к другим сервисам для получения идентификаторов или данных. К операциям, выполняемым после, — уведомления и индексирование. Примеры таких сервисов в спроектированной системе ленты новостей — сервис поглотителя, сервис потребителя, задания ETL и сервис бэкенда.
- Ведение журналов, мониторинг и оповещения должны работать при сбоях и аномальных событиях, которые имеют значение для системы.

17 Проектирование дашборда для топ-10 товаров Amazon по объему продаж

В ЭТОЙ ГЛАВЕ

- ✓ Масштабирование операции агрегирования для больших потоков данных
- ✓ Использование лямбда-архитектуры для быстрого получения приближенных результатов и медленного получения точных результатов
- ✓ Использование каппа-архитектуры как альтернативы лямбда-архитектуре
- ✓ Аппроксимация операций агрегирования для их ускорения

Аналитика — стандартная тема собеседований по проектированию систем. Вы всегда будете сохранять в журнале определенные сетевые запросы и взаимодействия с пользователем и выполнять аналитику на основании собранных данных.

Отображение верхних K позиций — стандартная функция многих дашбордов. На основании популярности (или непопулярности) товара можно принимать решения о его продвижении или отказе от товара. Такие решения могут быть нестандартными. Например, если товар непопулярен, можно решить вывести его из ассортимента, чтобы сэкономить затраты на его продажу, или же выделить дополнительные средства на рекламу для повышения его продаж.

Задача верхних K позиций — популярная тема при обсуждении аналитики и может стать отдельным вопросом на собеседовании. Задача может принимать бесконечное количество форм. Несколько примеров:

- товары с наибольшим (как в нашем вопросе) или наименьшим объемом продаж или доходом;
- чаще всего (или реже всего) просматриваемые товары в приложении интернет-магазина;
- приложения с наибольшим количеством загрузок в магазине приложений;
- самые просматриваемые видеоролики в приложении видеохостинга (например, YouTube);
- самые популярные (часто прослушиваемые) или наименее популярные песни в музыкальном приложении (например, Spotify);
- акции с наибольшим количеством операций на бирже (например, Robinhood или E*TRADE);
- наиболее часто пересылаемые посты в социальных сетях (например, твиты Twitter с наибольшим количеством ретвитов или посты в Instagram).

17.1. ТРЕБОВАНИЯ

Чтобы определить функциональные и нефункциональные требования, задайте себе несколько вопросов. Будем считать, что у нас есть доступ к датацентрам Amazon или любого другого приложения электронной коммерции, которое нас интересует.

- Как решать проблему равенства значений?

Высокая точность может быть не столь важна, так что можно выбрать любой элемент из набора с одинаковыми значениями.

- Какие временны́е интервалы нас интересуют?

Система должна быть способна агрегировать данные по заданным интервалам (часы, дни, недели, годы).

- Сценарии использования влияют на нужную точность (и другие требования, например масштабируемость). Как будет учитываться эта информация? Каковы требования к точности и согласованности/задержке?

Хороший вопрос. А что вы об этом думаете?

Определение точных объемов продаж и ранжирование в реальном времени потребует слишком больших затрат ресурсов. Если в системе используется

лямбда-архитектура, возможно, решение обеспечивает консистентность в конечном счете, что позволит узнать примерный объем продаж и рейтинг за несколько последних часов и точную информацию за предшествующие периоды.

Также можно пожертвовать точностью ради повышения масштабируемости, снижения затрат и сложности и упрощения сопровождаемости. Ожидается, что список верхних *K* позиций за конкретный период будет определяться как минимум через несколько часов после истечения этого периода, так что консистентность проблемой не является.

Низкая задержка тоже не является проблемой. Ожидается, что генерация списка займет несколько минут.

- Интересуют ли нас верхние *K* позиций (или только топ-10 товаров) либо произвольное количество товаров?

По аналогии с предыдущим вопросом, будет ли приемлемым решение, вычисляющее примерный объем продаж и рейтинг для топ-10 продуктов за несколько последних часов и объем продаж и рейтинг для произвольного количества продуктов за предшествующие периоды (теоретически до нескольких лет)? Также желательно, чтобы решение могло выводить более 10 продуктов.

- Нужно ли выводить объемы продаж для списка верхних *K* позиций или достаточно ранжирования товаров в этом списке?

В системе будут выводиться и рейтинг, и объем продаж. На первый взгляд вопрос может показаться искусственным, но если вы можете обойтись без вывода некоторых данных, это может существенно упростить дизайн.

- Нужно ли учитывать события, происходящие после продажи? Клиент может запросить возврат средств или обменять купленный товар на другой, либо товар может быть отозван производителем.

Это хороший вопрос, который демонстрирует опыт кандидата и его внимание к деталям. Нас будут интересовать только события продаж, а дальнейшее (например, споры или отзывы товаров) будем игнорировать.

- Обсудим требования к масштабированию. Какова частота транзакций сделок? С какой частотой выполняются запросы к дашборду верхних K позиций? Сколько товаров в ассортименте?

Допустим, происходит 10 миллиардов событий продаж в день (то есть высокий трафик продажных транзакций). При 1 Кбайт на событие интенсивность записи составит 10 Тбайт/день. Дашборд будет виден только сотрудникам, так что частота запросов будет достаточно низкой. Допустим, что в ассортименте около 1 миллиона товаров.

Другие нефункциональные требования отсутствуют. Высокая доступность или низкая задержка (и соответствующая сложность, которую они добавляют в дизайн системы) не требуется.

17.2. ИСХОДНЫЙ ВАРИАНТ

Первая мысль — регистрировать события в распределенном хранилище (таком, как HDFS или Elasticsearch) и выполнять запрос MapReduce, Spark или Elasticsearch каждый раз, когда потребуется вычислить список верхних *K* позиций за определенный период. Однако такой подход требует слишком высоких вычислительных затрат и много времени. Вычисление верхних *K* позиций за конкретный месяц или год может занять несколько часов или дней.

Если журналы событий продаж нужны только для генерации этого списка, обеспечивать их хранение за многие месяцы или годы слишком расточительно. При сохранении миллионов запросов в секунду за год накопится несколько петабайт данных. Возможно, вы решите хранить необработанные события за несколько месяцев или лет для разных целей, включая разрешение споров с клиентами и возврат средств, а также для диагностики или нормативно-правового соответствия. Однако этого периода удержания данных может быть недостаточно для генерации необходимого списка верхних *K* позиций.

Перед вычислением списков верхних *K* позиций необходимо провести предварительную обработку данных. Следует периодически выполнять агрегирование и подсчитывать продажи товаров с группировкой по часам, дням, неделям, месяцам и годам. После этого, когда вам понадобится список верхних *K* позиций, можно будет выполнить следующие действия:

1. Суммировать значения групп в зависимости от требуемого периода. Например, если вам понадобится список верхних K позиций за период в один месяц, вы просто используете группу этого месяца. Если вас интересует конкретный период за 3 месяца, суммируйте значения групп месяцев этого периода. Такой подход позволит сэкономить память за счет удаления событий после суммирования.
2. Отсортировать полученные суммы для получения списка верхних *K* позиций.

Необходимость хранения групп объясняется тем, что продажи могут быть очень неравномерными. В крайнем случае товар A может иметь 1 миллион продаж за конкретный час конкретного года и 0 продаж в остальное время этого года, тогда как продажи любых других товаров могут в сумме давать много меньше 1 миллиона продаж за этот год. В этом случае товар A будет находиться в списке верхних *K* позиций любого периода, включающего указанный час.

Оставшаяся часть этой главы посвящена масштабированному выполнению этих операций в распределенной системе.

17.3. ИСХОДНАЯ ВЫСОКОУРОВНЕВАЯ АРХИТЕКТУРА

Начнем с *лямбда-архитектуры* — стратегии обработки больших объемов данных с применением как пакетных, так и потоковых методов (см. https://www.databricks.com/glossary/lambda-architecture или https://www.snowlake.com/guides/lambda-architecture). На рис. 17.1 лямбда-архитектура состоит из двух пайплайнов параллельной обработки данных и сервисного слоя, объединяющего результаты этих двух пайплайнов:

1. Потоковый слой/пайплайн, который в реальном времени поглощает события от всех датацентров, где происходят транзакции продаж, и использует алгоритм для вычисления объемов продаж и рейтингов самых популярных товаров.
2. Пакетный слой, или пакетный пайплайн, выполняемый периодически (каждый час, день, неделю или год) для вычисления точных объемов продаж и рейтингов. Чтобы пользователи видели точные данные по мере того, как они становятся доступными, в задание ETL пакетного пайплайна можно включить задачу перезаписи результатов потокового пайплайна результатами пакетного пайплайна, когда они появятся.

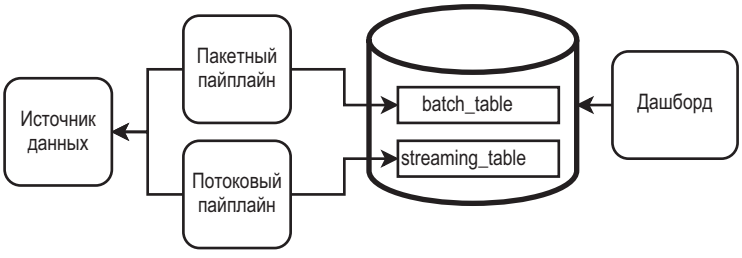

Рис. 17.1. Высокоуровневый эскиз лямбда-архитектуры. Стрелками обозначено направление запросов. Данные проходят через параллельные пайплайны, потоковый и пакетный. Каждый пайплайн записывает результаты своей работы в таблицу базы данных. Потоковый пайплайн записывает в streaming_table, тогда как пакетный пайплайн записывает в batch_table. Дашборд объединяет данные из streaming_table и batch_table, чтобы сгенерировать список верхних *K* позиций

Следуя принципам событийной архитектуры EDA (Event Driven Architecture), сервис продаж на бэкенде отправляет в топик Kafka события, которые могут использоваться для всей последующей аналитики (например, дашборда с верхними *K* позициями).

17.4. СЕРВИС АГРЕГИРОВАНИЯ

Первая возможная оптимизация лямбда-архитектуры — агрегирование событий продаж и отправка событий сводных данных в потоковые и пакетные пайплайны. Агрегирование может сократить размеры кластеров как потоковых, так и пакетных пайплайнов. Более детализированная исходная архитектура представлена на рис. 17.2. Потоковые и пакетные пайплайны записывают в реляционную СУБД (SQL), из которой дашборд может запрашивать данные с низкой задержкой. Если ваши потребности ограничиваются простой выборкой «ключ — значение», также можно воспользоваться Redis, но скорее всего, для дашборда и будущих сервисов вам дополнительно понадобятся операции фильтрации и агрегирования.

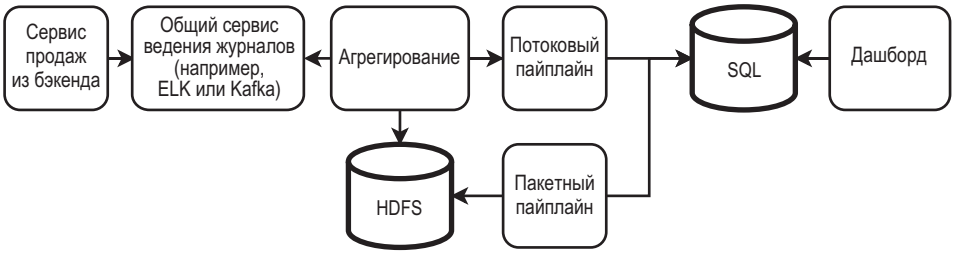

Рис. 17.2. Лямбда-архитектура, с исходным сервисом агрегирования, а также потоковыми и пакетными пайплайнами. Стрелками обозначено направление запросов. Сервис продаж из бэкенда направляет события (включая события продаж) в общий сервис ведения журналов, который является источником данных для дашборда. Сервис агрегирования потребляет события продаж из общего сервиса ведения журналов, агрегирует их и направляет агрегированные события в потоковый пайплайн и HDFS. Пакетный пайплайн вычисляет значения быстрее, но менее точно, чем пакетный пайплайн, и записывает их в таблицу SQL speed_table. Дашборд использует комбинацию данных из batch_table и speed_table для генерации списка верхних K позиций

> **ПРИМЕЧАНИЕ** Событийная архитектура (EDA) использует события для инициирования и взаимодействия между слабосвязанными сервисами (https://aws.amazon.com/event-driven-architecture/). За дополнительной информацией обращайтесь к другим источникам, например разделу 5.1 или с. 295 книги Артура Эйсмонта «Web Scalability for Startup Engineers».

Агрегирование, а также его плюсы и минусы обсуждались в разделе 4.5. Сервис агрегирования состоит из кластера producer-хостов топика Kafka, который регистрирует события продаж, агрегирует события и записывает агрегированные события в HDFS (через Kafka) и в потоковый пайплайн.

17.4.1. Агрегирование по идентификатору товара

Необработанное событие продаж может содержать такие поля, как (timestamp, product ID), тогда как агрегированное событие может принять форму (product_id, start_time, end_time, count, aggregation_host_id). События можно агрегировать благодаря тому, что точные метки времени не важны. Если важны определенные интервалы (например, в пределах часа), можно проверить, что пары (start_time, end_time) всегда принадлежат одному часу. Например, пара (0100, 0110) проходит проверку, а пара (0155, 0205) — нет.

17.4.2. Сопоставление идентификаторов хостов с идентификаторами продуктов

Сервис агрегирования может выполнить секционирование по идентификатору товара, чтобы каждый хост отвечал за агрегирование определенного набора идентификаторов. Для простоты можно вручную поддерживать сопоставление (host ID, product ID). Существуют разные варианты реализации этой конфигурации, в числе которых:

1. Файл конфигурации, включенный в исходный код сервиса. При каждом изменении файла придется перезапускать весь кластер.
2. Хранение файла конфигурации в общем хранилище объектов. Каждый хост сервиса читает этот файл при запуске и сохраняет в памяти идентификаторы товаров, за которые он отвечает. Сервису также понадобится конечная точка для обновления идентификаторов его товаров. При изменении файла можно обратиться с вызовом к этой конечной точке с хостов, которые потребляют разные идентификаторы товаров.
3. Хранение карты в таблице базы данных SQL или Redis.
4. Паттерн sidecar, в котором хост отправляет запрос на загрузку данных sidecar-контейнеру. Контейнер загружает событие для соответствующих идентификаторов товаров и возвращает его хосту.

Обычно мы будем выбирать вариант 2 или 4, чтобы не приходилось перезапускать весь кластер для каждого изменения конфигурации. Мы выбираем файл вместо базы данных по следующим причинам:

- Формат файлов конфигурации (например, YAML или JSON) легко парсить прямо в структуру данных хеш-карты. Чтобы добиться того же эффекта с таблицей базы данных, потребуется намного больше кода. Придется программировать в ORM-фреймворке, кодировать запрос к базе данных и объекту доступа к данным и сопоставлять объект доступа к данным с хеш-картой.
- Скорее всего, количество хостов не превысит нескольких сотен или нескольких тысяч, так что файл конфигурации будет совсем небольшим. Каждый

хост сможет загрузить весь файл, и решение с низкой задержкой чтения из базы данных не понадобится.

- Конфигурация будет изменяться слишком редко, чтобы оправдать затраты на использование такой базы данных, как SQL или Redis.

17.4.3. Хранение меток времени

Если нужно где-то сохранить точную метку времени, этой задачей будет заниматься сервис продаж, а не сервис аналитики или сервис товаров-лидеров. Необходимо обеспечить разделение обязанностей. Для событий продаж будут существовать и другие пайплайны аналитики помимо сервиса товаров-лидеров. Нужно обеспечить полную свободу разработки и вывода из эксплуатации этих пайплайнов безотносительно других сервисов. Иначе говоря, нужно тщательно продумать, должны ли другие сервисы зависеть от этих сервисов аналитики.

17.4.4. Процесс агрегирования на хосте

Хост агрегирования содержит хеш-таблицу, ключом которой является идентификатор товара, а значением — счетчик продаж. Он также создает контрольные точки в потребляемом топике Kafka, записывая конечные точки в Redis. Контрольные точки состоят из идентификаторов агрегируемых событий. Количество хостов сервиса агрегирования может превышать количество секций в топике Kafka, хотя это маловероятно, так как агрегирование — простая и быстрая операция. Каждый хост многократно:

1) потребляет событие из топика;
2) обновляет свою хеш-таблицу.

Хост агрегирования может выгружать на диск хеш-таблицу с заданной периодичностью или при исчерпании памяти (в зависимости от того, что произойдет раньше). Возможная реализация процесса выгрузки выглядит так:

1. Отправить агрегированные события в топик Kafka, которому можно присвоить, например, имя Flush. Если агрегированных данных немного (например, несколько мегабайт), их можно записать как одно событие, состоящее из списка кортежей агрегированных идентификаторов товаров с полями (*идентификатор_товара*, *метка_начала*, *метка_конца*, *объем_продаж*) — например, [(123, 1620540831, 1620545831, 20), (152, 1620540731, 1620545831, 18), . . .].
2. При использовании отслеживания измененных данных (CDC, см. раздел 5.3) у каждого приемника имеется потребитель, который потребляет событие и записывает в приемник:
   a) записывает агрегированные события в HDFS;

b) записывает в Redis кортеж контрольной точки со статусом complete (завершен) (например, {“hdfs”: “1620540831, complete”});

c) повторяет шаги 2a — 2в для потокового пайплайна.

Если топика Kafka Flush не будет и на хосте потребителя произойдет сбой во время записи агрегированного события в приемник, сервису агрегирования придется заново агрегировать эти события.

Для чего нужно записывать две контрольные точки? Это всего лишь один из возможных алгоритмов обеспечения консистентности. Если на шаге 1 на хосте произойдет сбой, другой хост может потребить событие выгрузки и выполнить запись. Если на хосте произойдет сбой на шаге 2a, запись в HDFS может пройти успешно или завершиться ошибкой. Другой хост может прочитать данные из HDFS, чтобы проверить, была ли запись успешно завершена или попытку необходимо повторить. Чтение из HDFS является затратной операцией. Так как сбой хоста — событие редкое, эта затратная операция также будет редкой. Если необходим этот затратный механизм восстановления после сбоев, его можно реализовать в виде периодической операции, которая читает все контрольные точки обработки за период от последней минуты до нескольких минут.

Сам механизм восстановления после сбоев должен быть идемпотентным на случай, если в нем произойдет сбой во время выполнения и его нужно будет повторить.

Также следует уделить внимание отказоустойчивости. Любая операция записи может завершиться неудачей. На любом хосте в сервисе агрегирования, сервисе Redis, кластере HDFS или потоковом пайплайне в любой момент может произойти сбой. Могут возникнуть сетевые проблемы, прерывающие запросы записи к любому хосту в сервисе. Даже если ответ события записи содержит код 200, могла произойти скрытая ошибка. Это приведет к тому, что три сервиса окажутся в рассогласованном состоянии. Следовательно, необходимо записать отдельную контрольную точку для HDFS и потокового пайплайна. Событие записи должно иметь идентификатор, чтобы сервисы приемника могли выполнить дедупликацию при необходимости.

Как в ситуациях, в которых событие необходимо записать в несколько сервисов, предотвратить рассогласование?

1. Контрольная точка после каждой записи в каждый сервис, как описано выше.
2. Если в требованиях указано, что несогласованность приемлема, можно не делать ничего. Например, можно допустить некоторую неточность в потоковом пайплайне, но пакетный пайплайн должен быть точным.
3. Периодический аудит (так называемый супервизор). Если значения не согласуются, несогласованные результаты отбрасываются и соответствующие данные обрабатываются повторно.

4. Использовать технологии распределенных транзакций — 2PC, сага, CDC или супервизор транзакций. Они рассматриваются в главе 4 и приложении Г.

Как упоминалось в разделе 4.5, у агрегирования есть оборотная сторона: результаты реального времени задерживаются на время, необходимое для агрегирования и выгрузки. Агрегирование может быть неподходящим, если дашборд требует обновлений с низкой задержкой.

17.5. ПАКЕТНЫЙ ПАЙПЛАЙН

Пакетный пайплайн концептуально проще потокового пайплайна, поэтому мы с него и начнем.

На рис. 17.3 показана упрощенная потоковая диаграмма пакетных пайплайнов. Пакетный пайплайн состоит из серии задач агрегирования/свертки за счет повышения интервалов. Свертка осуществляется по часам, затем по дням, затем по неделям и, наконец, по месяцам и годам. Если у вас имеется миллион идентификаторов товаров:

1. Свертка по часам генерирует 24 миллиона строк в день, или 168 миллионов строк в неделю.
2. Свертка по месяцам генерирует 28–31 миллион строк в месяц, или 168 миллионов строк в год.
3. Свертка по дням генерирует 7 миллионов строк в неделю, или 364 миллиона строк в год.

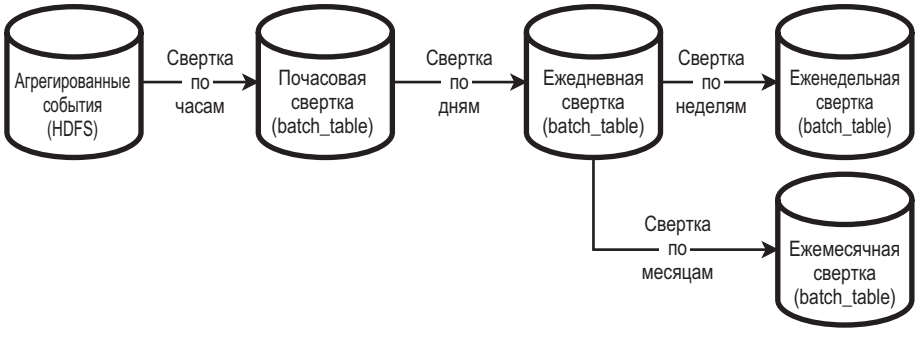

Рис. 17.3. Структурная схема задач свертки в пакетном пайплайне. Задание свертки последовательно выполняет свертку с увеличением временных интервалов для сокращения количества строк, обрабатываемых на каждой стадии

Оценим требования к объему хранилища данных. Четыреста миллионов строк, каждая из которых содержит десять 64-битовых столбцов, занимают 32 Гбайт. Такой объем данных легко помещается на одном хосте. Заданию почасовой свертки

может потребоваться обрабатывать миллиарды событий продажи, поэтому оно может использовать запрос Hive для чтения из HDFS и последующей записи полученных счетчиков в таблицу SQL batch_table. Свертки для других интервалов используют значительное сокращение количества строк от почасовой свертки, и им необходимо читать и записывать данные только в таблицу SQL batch_table.

В каждой из этих сверток можно упорядочить счетчики по убыванию и записать верхние K позиций (или возможно, $K \times 2$ для гибкости) строк в базу данных SQL для вывода на дашборде.

На рис. 17.4 представлен простой направленный ациклический граф (DAG) ETL нашего пакетного пайплайна (то есть одно задание свертки). Для каждой свертки будет существовать один DAG (то есть всего четыре таких DAG). DAG ETL состоит из следующих четырех задач (третья и четвертая задачи являются одноуровневыми). Мы используем терминологию Airflow:

1. Для всех сверток, кроме почасовых, понадобится задача для проверки того, что выполнение зависимых сверток завершилось успешно. Задача может также проверить, что необходимые данные HDFS или данные SQL доступны, но для этого потребуются затратные запросы к базам данных.
2. Выполните запрос Hive или SQL, который отсортирует счетчики по убыванию и запишет результаты в batch_table.
3. Удалите соответствующие строки из speed_table. Эта задача отделена от задачи 2, поскольку задача 3 может быть повторно выполнена без повторного выполнения задачи 2. Если в задаче 3 произойдет сбой при попытке удаления строк, удаление повторяется без повторного выполнения затратного запроса Hive или SQL из шага 2.
4. Создайте или сгенерируйте заново подходящие списки верхних K позиций, используя новые строки batch_table. Как обсуждалось в разделе 17.5, эти списки верхних K позиций с большой вероятностью будут сгенерированы с использованием как точных данных batch_table, так и неточных данных speed_table, так что они будут регенерироваться только с batch_table. Эта задача не требует высоких затрат, но может повторно выполняться отдельно в случае сбоя, поэтому мы и выделяем ее в отдельную задачу.

Что касается задачи 1, ежедневная свертка может происходить только в том случае, если все почасовые свертки-зависимости были записаны в HDFS; то же самое относится к еженедельным и ежемесячным сверткам. Одна ежедневная свертка зависит от 24 почасовых сверток, одна еженедельная свертка зависит от 7 ежедневных сверток, и одна ежемесячная свертка зависит от 28–30 ежедневных сверток (в разные месяцы). При работе с Airflow можно использовать экземпляры ExternalTaskSensor (https://airlow.apache.org/docs/apache-airlow/stable/howto/operator/external_task_sensor.html#externaltasksensor) с соответствующими значениями параметра execution_date в ежедневных, еженедельных

и ежемесячных HAГ для проверки того, что свертки-зависимости завершены успешно.

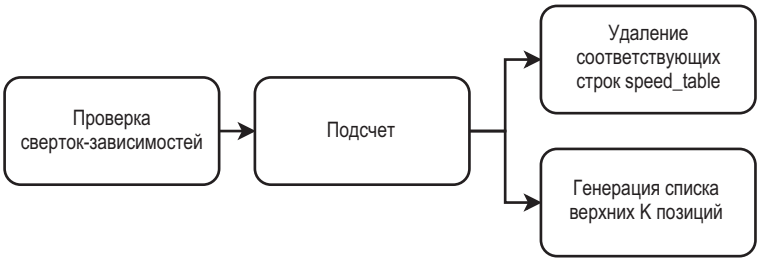

Рис. 17.4. DAG ETL для одного задания свертки. Задачи-компоненты должны проверить, что свертки-зависимости завершены, выполнить свертку/подсчет и сохранить данные в SQL, после чего удалить соответствующие строки speed_table, которые больше не нужны

17.6. ПОТОКОВЫЙ ПАЙПЛАЙН

Выполнение пакетного задания может занять много часов, что повлияет на свертки для всех интервалов. Например, запрос Hive для последнего почасового задания может занять 30 минут, так что следующие свертки (и как следствие, их списки верхних *K* позиций) будут недоступны:

- список верхних *K* позиций за этот час;
- список верхних *K* позиций за день, содержащий этот час;
- списки верхних *K* позиций за неделю и месяц, содержащие этот день.

Цель потокового пайплайна — получить счетчики (и списки верхних *K* позиций), еще не предоставленные пакетным пайплайном. Потоковый пайплайн должен вычислять эти счетчики намного быстрее, чем пакетный; при этом могут использоваться приближенные методы.

Следующий шаг после исходного агрегирования — вычисление итоговых счетчиков и сортировка их по убыванию, в результате чего получается список верхних *K* позиций. В этом разделе мы сначала найдем решение для одного хоста, а затем обеспечим его горизонтальную масштабируемость.

17.6.1. Хеш-таблица и двоичная куча max-heap с одним хостом

В первой версии мы используем хеш-таблицу, отсортированную по счетчикам частот с использованием двоичной кучи max-heap размером K. В листинге 17.1 приведен пример функции Golang, в которой используется этот подход.

Листинг 17.1. Пример функции Golang для вычисления списка верхних K позиций

```
type HeavyHitter struct {
  identifier string
  frequency int
}

func topK(events []String, int k) (HeavyHitter) {
  frequencyTable := make(map[string]int)
  for _, event := range events {
    value := frequencyTable[event]
    if value == 0 {
      frequencyTable[event] = 1
    } else {
      frequencyTable[event] = value + 1
    }
  }

  pq = make(PriorityQueue, k)
  i := 0
  for key, element := range frequencyTable {
    pq[i++] = &HeavyHitter{
      identifier: key,
      frequency: element
    }
    if pq.Len() > k {
      pq.Pop(&pq).(*HeavyHitter)
    }
  }

  /*
   * Записывает содержимое кучи в приемник.
   * Здесь они просто возвращаются в массиве.
   */
  var result [k]HeavyHitter
  i := 0
  for pq.Len() > 0 {
    result[i++] = pq.Pop(&pq).(*HeavyHitter)
  }
  return result
}
```

В нашей системе несколько экземпляров функции могут выполняться параллельно для разных временны́х групп (то есть часа, дня, недели, месяца и года). В конце каждого периода можно сохранить содержимое кучи max-heap, обнулить счетчики и начать отсчет за новый период.

17.6.2. Горизонтальное масштабирование по нескольким хостам и многоуровневое агрегирование

На рис. 17.5 представлено горизонтальное масштабирование по нескольким хостам и многоуровневое агрегирование. Два хоста в среднем столбце суммируют

счетчики (товар, час) от предшествующих хостов в левом столбце, тогда как кучи max-heap в правом столбце агрегируют счетчики (товар, час) от предшествующих хостов в среднем столбце.

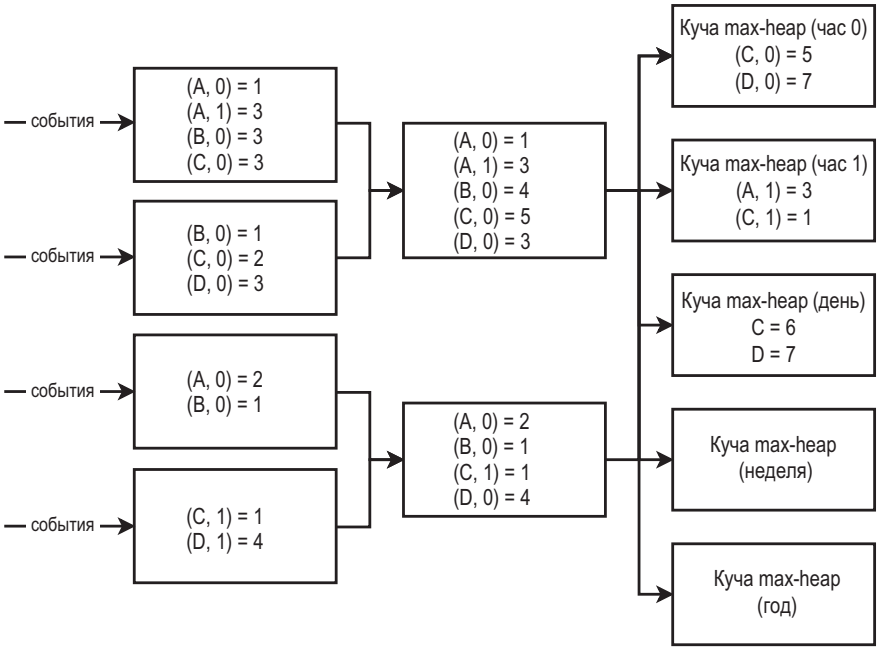

Рис. 17.5. Если трафик к хосту итоговой хеш-таблицы будет слишком высоким, для потокового пайплайна можно воспользоваться многоуровневой структурой. Для краткости ключ отображается в формате (товар, час). Например, (A, 0) обозначает товар A в час 0. Последний уровень хостов может содержать двоичные кучи max-heap, по одной для каждого интервала свертки. Эта структура очень похожа на многоуровневый сервис агрегирования, рассмотренный в разделе 4.5.2. У каждого хоста имеется связанный топик Kafka, который не показан на схеме

Агрегирование

Обратите внимание: агрегирование выполняется по комбинации идентификатора товара и метки времени. Прежде чем читать дальше, подумайте почему.

В такой структуре между первым уровнем хостов и хостом итоговой хеш-таблицы могут добавляться дополнительные уровни, чтобы ни один хост не получал больше трафика, чем он может обработать. Таким образом, сложность реализации мгногоуровневого сервиса агрегирования просто смещается с сервиса агрегирования на потоковый пайплайн. Решение создает задержку, как было описано в разделе 4.5.2. Также выполняется секционирование по схеме,

описанной в разделе 4.5.3 и показанной на рис. 4.6. Обратите внимание на то, что сказано в этом разделе относительно проблемы горячих шардов. Мы используем секционирование по идентификатору товара, но также можно секционировать по метке времени события продаж.

Почему мы агрегируем по комбинации идентификатора товара и метки времени? Потому что список верхних *K* позиций имеет определенный период с начальным и конечным временем. Необходимо убедиться, что каждое событие продаж агрегируется в правильных временных диапазонах. Например, событие продаж, произошедшее в момент 2023-01-01 10:08 UTC, должно агрегироваться в следующих диапазонах:

- Час [2023-01-01 10:08 UTC, 2023-01-01 11:00 UTC).
- День [2023-01-01, 2023-01-02).
- Неделя [2022-12-28 00:00 UTC, 2023-01-05 00:00 UTC). Оба дня — 2022-12-28 и 2023-01-05 — являются понедельниками.
- Месяц [2023-01-01, 2013-02-01).
- Год [2023, 2024).

В одном из решений агрегирование проводится по наименьшему периоду (то есть часу). Предполагается, что прохождение любого события по всем уровням занимает всего несколько секунд, поэтому возраст любого ключа в кластере вряд ли превысит час. Каждый идентификатор товара имеет собственный ключ. С присоединением диапазона часов к каждому ключу количество ключей также вряд ли превысит удвоенное количество идентификаторов товаров.

Через одну минуту после окончания периода — например, 2023-01-01 11:01 UTC для [2023-01-01 10:08 UTC, 2023-01-01 11:00 UTC) или 2023-01-02 00:01 UTC для [2023-01-01, 2023-01-02) — соответствующий хост итогового уровня (который мы будем называть *итоговым хостом*) может записать свою кучу в таблицу SQL speed_table, после чего дашборд будет готов вывести соответствующий список верхних *K* позиций за этот период. В отдельных случаях прохождение события по всем уровням может занять более минуты, тогда итоговые хосты могут просто записать обновленные кучи в speed_table. Можно установить для итоговых хостов период удержания старых ключей агрегирования в несколько часов или дней, по истечении которого хосты смогут удалить их.

Альтернатива минутному ожиданию — реализация системы, которая отслеживает события, проходящие через хосты, и инициирует запись куч в speed_table итоговыми хостами только после того, как все актуальные события достигнут итоговых хостов. Однако такое решение может быть слишком сложным, к тому же оно не позволяет дашборду выводить приближенную информацию, пока не будут полностью обработаны все события.

17.7. АППРОКСИМАЦИЯ

Для снижения задержки можно ограничить количество уровней в сервисе агрегирования. На рис. 17.6 представлен пример такого дизайна. В нем используются

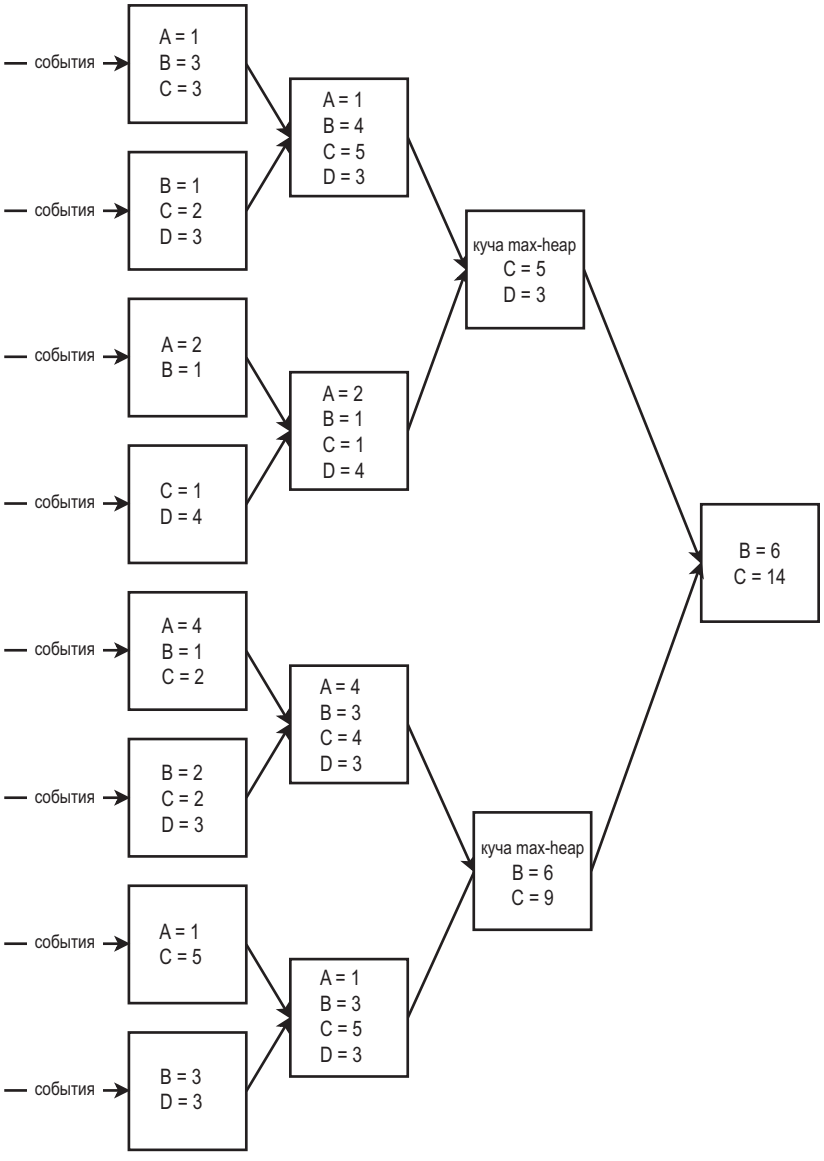

Рис. 17.6. Многоуровневая структура с кучами max-heap. Агрегирование выполняется быстрее, но менее точно. Для краткости на схеме не представлены временные группы

уровни, состоящие только из куч max-heap. Такой подход жертвует точностью ради ускорения обновлений и снижения затрат. Также можно воспользоваться пакетным пайплайном для более медленного, но более точного агрегирования.

Почему кучи max-heap размещаются на отдельных хостах? Это сделано для того, чтобы упростить предоставление новых хостов при масштабировании кластера. Как отмечалось в разделе 3.1, система считается масштабируемой, если она легко масштабируется по возрастанию и убыванию. Для хостов хеш-таблиц и хоста кучи max-heap могут быть отдельные образы Docker, поскольку количество хостов хеш-таблиц может часто изменяться, тогда как активный хост кучи max-heap (и его реплики) может быть только один.

Тем не менее список верхних *K* позиций, полученный с помощью этой структуры, может быть неточным. Нельзя поддерживать кучу max-heap на каждом хосте и просто выполнять слияние пирамид, поскольку в этом случае итоговая куча может не содержать верхние *K* товаров. Например, если хост 1 содержит хеш-таблицу {A: 7, B: 6, C: 5}, а хост B — хеш-таблицу {A: 2, B: 4, C: 5}, тогда как куча max-heap имеет размер 2, куча max-heap хоста 1 будет содержать {A: 7, B: 6}, а куча max-heap хоста 2 — {B: 4, C: 5}. Итоговая объединенная куча max-heap будет содержать {A: 7, B: 10}, и в результате C ошибочно не войдет в верхнюю двойку. Корректная куча max-heap должна иметь вид {B: 10, C: 11}.

17.7.1. Алгоритм Count-min sketch

Рассмотренные примеры требуют большого объема памяти на каждом хосте для хеш-таблицы, размер которой соответствует количеству товаров (в нашем случае примерно 1 миллион). Можно воспользоваться приближенными вычислениями, жертвуя точностью в целях снижения затрат памяти.

Для этого хорошо подойдет алгоритм аппроксимации Count-min sketch. Его можно представить как двумерную (2D) таблицу, обладающую шириной и высотой. Ширина обычно составляет несколько тысяч, тогда как высота невелика и представляет количество хеш-функций (например, 5). Вывод каждой хеш-функции связывается с шириной. При поступлении нового элемента к нему применяется каждая хеш-функция и увеличивается соответствующая ячейка.

Рассмотрим пример использования Count-min sketch с простой последовательностью «A C B C C». C — самая частая буква, встречающаяся три раза. В табл. 17.1–17.5 показано изменение Count-min sketch. Хешируемое значение на каждом шаге выделено жирным шрифтом.

1. Хешировать первую букву A каждой из пяти хеш-функций. В табл. 17.1 показано, что все хеш-функции преобразуют A в новое значение.

Таблица 17.1. Таблица Count-min sketch после добавления буквы A

1					
	1				
				1	
	1				
	1				

2. Хешировать вторую букву C. В табл. 17.2 показано, что первые четыре хеш-функции преобразуют C в значение, отличное от значения A. С пятой хеш-функцией возникает коллизия. Хешированные значения A и C совпадают, поэтому это значение увеличивается.

Таблица 17.2. Таблица Count-min sketch после добавления букв A C

1				1	
	1		1		
1				1	
	1		1		
	2 (коллизия)				

3. Хешировать третью букву B. Из табл. 17.3 видно, что с четвертой и пятой функцией возникают коллизии.

Таблица 17.3. Таблица Count-min sketch после добавления букв A C B

1		1		1	
	1		1		1
1	1			1	
	2 (коллизия)		1		
	3 (коллизия)				

4. Хешировать четвертую букву C. Из табл. 17.4 видно, что коллизия возникает только с пятой функцией.

Таблица 17.4. Таблица Count-min sketch после добавления букв A C B C

1		1		2	
	1		2		1
2	1			1	
	2		2		
	4 (коллизия)				

5. Хешировать пятую букву C. Операция идентична предыдущей. В табл. 17.5 приведена таблица Count-min sketch после всей последовательности A C B C C.

Таблица 17.5. Таблица Count-min sketch после добавления букв A C B C C

1		1		3	
	1		3		1
3	1			1	
	2		3		
	5 (коллизия)				

Чтобы найти элемент с наибольшим количеством вхождений, сначала следует найти максимумы по всем строкам {3, 3, 3, 3, 5}, а затем найти минимум среди этих максимумов «3». Чтобы найти элемент со вторым по порядку количеством вхождений, найдите второе сверху число в каждой строке {1, 1, 1, 2, 5}, а затем найдите минимум среди этих чисел «1» — и т. д. Определяя минимум, вы снижаете риск завышения оценки.

Существуют специальные формулы для вычисления ширины и высоты в зависимости от желаемой точности и вероятности ее достижения. Эта тема выходит за рамки книги.

Двумерный массив Count-min sketch заменяет хеш-таблицу из предыдущих решений. Вам все равно понадобится куча для хранения списка лидеров, но потенциально большая хеш-таблица заменяется двумерным массивом заранее определенного размера, который остается фиксированным независимо от размера набора данных.

17.8. ДАШБОРД С ЛЯМБДА-АРХИТЕКТУРОЙ

Дашборд может быть реализован в виде браузерного приложения, которое отправляет запрос GET к бэкенд-сервису, а тот, в свою очередь, отправляет запрос SQL (рис. 17.7). Используется пакетный пайплайн, который записывает в таблицу batch_table, и потоковый пайплайн, который записывает в таблицу speed_table, а дашборд строит список верхних K позиций по обеим таблицам.

Однако таблицы SQL не гарантируют определенного порядка, и операции фильтрации и сортировки batch_table и speed_table могут занимать секунды. Чтобы задержка 99-го процентиля составляла менее 1 секунды, требуется простой запрос SELECT к одному представлению, содержащему список рейтингов и счетчиков, которое мы будем называть представлением top_1000. Это представление можно построить выборкой верхних 1000 продуктов из speed_table и batch_table в каждый период. Оно также может содержать дополнительный

столбец, который указывает, принадлежит ли каждая строка speed_table или batch_table. Когда пользователь запрашивает список верхних K позиций для конкретного интервала, бэкенд может запросить у этого представления как можно больше данных из таблицы batch_table и заполнить пропуски из таблицы speed_table. Как упоминалось в разделе 4.10, браузерное приложение и бэкенд-сервис также могут кэшировать ответы на запросы.

Рис. 17.7. Дашборд имеет простую архитектуру: в него входит браузерное приложение, которое отправляет запросы GET к бэкенд-сервису, который, в свою очередь, отправляет запросы SQL. Функциональные требования браузерного приложения могут расти со временем, от простого вывода верхних 10 элементов списка за конкретный период (например, предыдущий месяц) до включения больших списков, больших периодов, фильтрации или агрегирования (процентили, средние, мода, максимум, минимум)

Упражнение

Составьте запрос SQL для представления top_1000.

17.9. КАППА-АРХИТЕКТУРА

Каппа-архитектура — паттерн программной архитектуры для обработки потоковых данных, выполнения как пакетной, так и потоковой обработки в одном технологическом стеке (https://hazelcast.com/glossary/kappa-architecture). Она использует неизменяемый журнал с поддержкой только добавления данных (например, Kafka) для сохранения входных данных, после чего выполняются операции потоковой обработки и сохранения информации в базе данных, откуда ее могут запрашивать пользователи.

В этом разделе мы сравним лямбда- и каппа-архитектуру и обсудим применение каппа-архитектуры для нашего дашборда.

17.9.1. Лямбда- и каппа-архитектура

Лямбда-архитектура сложна, поскольку как пакетный, так и потоковый слой требуют собственной кодовой базы и кластера вкупе со связанными расходами на их эксплуатацию, а также затрат на разработку, обслуживание, ведение журналов, мониторинг и оповещения.

Каппа-архитектура является упрощенной версией лямбда-архитектуры, в которой есть только потоковый слой и нет пакетного. Это похоже на выполнение

потоковой и пакетной обработки в одном технологическом стеке. Слой обслуживания предоставляет данные, вычисленные по потоковому слою. Все данные читаются и преобразуются непосредственно после их вставки в ядро обмена сообщениями и обработки потоковыми средствами. В результате каппа-архитектура подходит для обработки данных с низкой задержкой и почти в реальном времени, как в дашбордах или в мониторинге реального времени. Как и для потокового слоя лямбда-архитектуры, можно отказаться от максимальной точности ради производительности. Однако можно не соблюдать этот компромисс и вычислить точные данные.

Каппа-архитектура основана на предположении, что пакетные задания вообще не обязательны и для всех операций и требований обработки данных достаточно потоковой обработки. См. https://www.oreilly.com/radar/questioning-the-lambda-architecture/ и https://www.kai-waehner.de/blog/2021/09/23/real-time-kappa-architecture, где рассматриваются недостатки пакетной обработки, отсутствующие у потоковой обработки.

Кроме рассматриваемых по ссылкам, другим недостатком пакетных заданий по сравнению с потоковыми становится существенный рост затрат на разработку и текущую эсплуатацию, поскольку пакетное задание, использующее распределенную файловую систему (например, HDFS), обычно выполняется минимум несколько минут даже на небольшом объеме данных. Это объясняется большим размером блока HDFS (64 или 128 Мбайт по сравнению с 4 Кбайт для файловых систем UNIX), из-за отказа от низкой задержки ради высокой пропускной способности. С другой стороны, на выполнение потокового задания, обрабатывающего небольшой объем данных, может понадобиться всего несколько секунд.

Сбои пакетных заданий практически неизбежны на всем жизненном цикле, от разработки до тестирования и эксплуатации, а в случае сбоя пакетное задание придется выполнять заново. Один из популярных способов сократить время ожидания пакетного задания — разбить его на стадии. На этом принципе строятся НАГ Airflow. Вы как разработчик можете проектировать пакетные задания так, чтобы они не занимали более 30 минут или 1 часа каждое, однако разработчикам и группе эксплуатации все равно придется ожидать полчаса или час, чтобы узнать, успешно ли завершилось задание. Хорошее покрытие тестами сокращает проблемы в ходе эксплуатации, но не устраняет их.

В целом ошибки в пакетных заданиях обходятся дороже, чем в потоковых. В пакетных заданиях одна ошибка приводит к сбою всего задания, тогда как в потоковых она влияет только на обработку одного конкретного задания.

Другое преимущество каппа-архитектуры перед лямбда-архитектурой — ее относительная простота, поскольку используется всего один фреймворк обработки, тогда как лямбда-архитектуре могут потребоваться разные фреймворки для пакетных и потоковых пайплайнов. Для потоковой обработки могут использоваться такие фреймворки, как Redis, Kafka и Flink.

Одна из особенностей каппа-архитектуры заключается в том, что хранение большого объема данных на платформе потоковой передачи событий (такой, как Kafka) обходится дорого и не масштабируется за пределы нескольких петабайт — в отличие от системы HDFS, спроектированной для больших объемов. Kafka обеспечивает бесконечное удержание данных со *сжатием журналов* (https://kafka.apache.org/documentation/#compaction), так что топик Kafka экономит память, сохраняя лишь последнее значение для каждого ключа сообщения, и удаляет все более ранние значения для этого ключа. Другое решение основано на использовании хранилищ объектов (таких, как S3) для долгосрочного хранения редко используемых данных. В табл. 17.6 приведено сравнение лямбда- и каппа-архитектуры.

Таблица 17.6. Сравнение лямбда- и каппа-архитектуры

Лямбда	Каппа
Отдельные пакетные и потоковые пайплайны. Отдельные кластеры, кодовые базы и фреймворки обработки. Каждой подсистеме необходима собственная инфраструктура, мониторинг, ведение журналов и поддержка	Один пайплайн, кластер, кодовая база и фреймворк обработки
Пакетные пайплайны обеспечивают более высокую производительность при обработке больших объемов данных	Обработка больших объемов данных выполняется медленнее и дороже, чем в лямбда-архитектуре. Однако данные обрабатываются сразу же после поглощения, в отличие от пакетных заданий, которые выполняются по графику, что позволяет получить данные быстрее
Ошибка в пакетном задании может потребовать повторной обработки всех данных с самого начала	Ошибка в потоковом задании требует повторной обработки только текущей точки данных

17.9.2. Каппа-архитектура для дашборда

Каппа-архитектура для дашборда с верхними K позициями может использовать подход из раздела 17.3.2, когда все события продаж агрегируются по идентификатору продукта и диапазону времени. Сохранять события продаж в HDFS с последующим выполнением пакетного задания не нужно. Данные 1 миллиона товаров могут легко поместиться на одном хосте, но один хост не сможет поглотить 1 миллиард событий за день; потребуется многоуровневое агрегирование.

Серьезная ошибка может повлиять на множество событий, поэтому необходимо регистрировать и отслеживать ошибки, а также частоту их возникновения. Такие проблемы трудно диагностировать, как и перезапустить потоковый пайплайн при большом количестве событий, так что можно определить критическую частоту ошибок и остановить пайплайн (запретить потребителям

Kafka потреблять и обрабатывать события), если частота ошибок превысит критическое значение.

На рис. 17.8 изображена высокоуровневая каппа-архитектура системы. По сути, это лямбда-архитектура с рис. 17.2 без пакетного пайплайна и сервиса агрегирования.

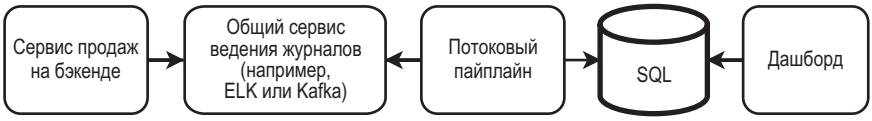

Рис. 17.8. Высокоуровневая система, использующая каппа-архитектуру. По сути, это лямбда-архитектура с рис. 17.2 без пакетного пайплайна и сервиса агрегирования

17.10. ВЕДЕНИЕ ЖУРНАЛОВ, МОНИТОРИНГ И ОПОВЕЩЕНИЯ

В дополнение к сказанному в разделе 2.5 отслеживать и отправлять оповещения необходимо в следующих случаях:

- Общая пакетная платформа ETL должна быть уже интегрирована с системами ведения журналов, мониторинга и оповещения. Вы будете получать оповещения об аномально долгом времени выполнения или сбоях задач, входящих в задание свертки.
- Задачи свертки записывают данные в таблицы HDFS. Средства мониторинга качества данных, описанные в главе 10, помогут выявить недействительные точки данных и отправлять оповещения.

17.11. ДРУГИЕ ВОЗМОЖНЫЕ ТЕМЫ ОБСУЖДЕНИЯ

- Используйте для создания списков другие параметры, например страну или город. Какие изменения нужно будет внести в дизайн, чтобы возвращать верхние K товаров по величине выручки (вместо объема продаж)? Как отслеживать верхние K товаров по изменениям в объемах продаж и/или выручки?
- Может быть полезно проводить поиск рейтинга и статистики по некоторым товарам с именами или описаниями, соответствующими заданным шаблонам. Для таких ситуаций можно спроектировать поисковую систему. Также можно обсудить программное использование списков верхних K позиций, например, сервисами машинного обучения и экспериментирования. Изначально мы исходили из того, что частота запросов невелика, а высокая доступность и низкая задержка не обязательны. Для программного использования эти предположения неприменимы, так как программные пользователи вводят новые нефункциональные требования.

- Может ли дашборд выводить приближенные значения для списков верхних *K* позиций, не дожидаясь завершения полного подсчета — или даже возникновения — событий?
- Подсчет продаж существенно усложняется из-за спорных ситуаций, например требования покупателем возврата средств или обмена. Должны ли данные продаж включать незавершенные споры? Нужно ли корректировать прошлые данные с учетом возврата, обмена или возмещения средств? Как пересчитывать события продаж в случае согласия или отказа в возврате средств или при обмене товара на тот же или другой товар?
- Компания может предоставлять гарантию на несколько лет, так что спор может возникнуть через несколько лет после продажи. Возможно, запросам к базам данных придется искать события продаж, произошедшие за годы до этого. Такие задания могут привести к исчерпанию свободной памяти. Это сложная задача, которую до сих пор пытаются решить многие инженеры.
- Возможны и такие радикальные события, как отзыв товара производителем. Например, приходится отзывать игрушку, поскольку неожиданно выяснилось, что она небезопасна для детей. Можно обсудить, следует ли корректировать объемы продаж при возникновении подобных проблем.
- Кроме повторной генерации списков верхних *K* позиций по описанным выше причинам, можно обобщить этот принцип для повторной генерации списков верхних *K* позиций при любых изменениях данных.
- Браузерное приложение выводит только список верхних *K* позиций. Можно расширить функциональные требования, например вывод трендов продаж или прогнозирование будущих продаж текущих или новых товаров.

17.12. ДОПОЛНИТЕЛЬНЫЕ ИСТОЧНИКИ

В этой главе использован материал презентации Top K Problem (Heavy Hitters) (https://youtu.be/kx-XDoPjoHw) на YouTube-канале System Design Interview, автор — Михаил Смарщок (Mikhail Smarshchok).

ИТОГИ

- Если точные крупномасштабные операции агрегирования занимают слишком много времени, можно запустить параллельный потоковый пайплайн, использующий методы приближенных вычислений. Такое решение жертвует точностью ради скорости. Параллельное выполнение быстрого неточного пайплайна с медленным точным пайплайном называется лямбда-архитектурой.

- Один из этапов крупномасштабного агрегирования — разбиение по ключу, по которому далее будет выполняться агрегирование.
- Данные, не связанные напрямую с агрегированием, должны храниться в другом хранилище, чтобы их могли использовать другие сервисы.
- Создание контрольных точек — один из возможных методов выполнения распределенных транзакций, включающих приемники как с низкозатратными операциями чтения (например, Redis), так и затратными операциями чтения (например, HDFS).
- Для приближенных крупномасштабных операций агрегирования можно использовать комбинацию куч max-heap и многоуровневого горизонтального масштабирования.
- Count-min sketch — метод приближенного подсчета.
- Для обработки больших потоков данных можно использовать каппа- или лямбда-архитектуру.

А Монолитные архитектуры и микросервисы

В этом приложении сравниваются монолитные архитектуры и микросервисы. Личный опыт автора показывает, что многие источники описывают преимущества микросервисов перед монолитной архитектурой, но не упоминают о недостатках, поэтому мы обсудим их здесь. Термины «сервис» и «микросервис» будут использоваться как синонимы.

Микросервисная архитектура — это способ построения программной системы в виде совокупности слабо связанных и независимо разрабатываемых, развертываемых и масштабируемых сервисов. Монолиты проектируются, разрабатываются и развертываются как единое целое.

А.1. ПРЕИМУЩЕСТВА МОНОЛИТНЫХ АРХИТЕКТУР

В табл. А.1 сравниваются преимущества монолитов над сервисами.

Таблица А.1. Преимущества монолитов над сервисами

Монолит	Сервис
Быстрота и простота разработки на начальной стадии, так как весь код объединен в одно приложение	Разработчики должны реализовать сериализацию и десериализацию в каждом сервисе и обрабатывать запросы и ответы между сервисами. Перед началом разработки необходимо определить границы между сервисами, но выбранные границы могут оказаться неверными. Перерабатывать сервисы для изменения границ обычно нецелесообразно
Использование одной базы данных означает, что решение требует меньшего объема хранилища, но у этого преимущества есть и оборотная сторона	Каждый сервис должен иметь собственную базу данных, что может приводить к дублированию данных и общему повышению требований к пространству хранения

Таблица А.1 (*окончание*)

Монолит	Сервис
С одной базой данных и меньшим количеством мест хранения данных проще обеспечить требования норм конфиденциальности данных	Данные разбросаны по разным местам, из-за чего труднее обеспечить соблюдение норм конфиденциальности данных в рамках организации
Возможное упрощение отладки. Разработчик может установить точку прерывания для просмотра стека вызовов в любой строке кода и понять всю логику происходящего в этой строке	Средства распределенной трассировки (такие, как Jaegar или Zipkin) используются для анализа разветвления запросов, но они не сообщают многие подробности, например стек вызовов функций сервисов, участвующих в запросе. Отладка нескольких сервисов обычно сложнее, чем отладка монолита или отдельного сервиса
К сказанному выше: возможность простого просмотра всего кода в одном месте и трассировка вызовов функции упрощает понимание приложения/системы в целом по сравнению с сервисной архитектурой	API сервиса работает по принципу «черного ящика». Хотя разработчику не требуется разбираться в подробностях API, что упрощает его использование, может быть труднее понять неочевидные особенности многих систем
Дешевизна эксплуатации и более высокая производительность. Вся обработка выполняется в памяти одного хоста, так что данные не передаются между хостами, что было бы намного медленнее и намного дороже	Система сервисов, между которыми передаются большие объемы данных, может создать очень высокие затраты при передачах данных между хостами и датацентрами. Пример того, как Amazon Prime Video удалось сократить инфраструктурные затраты системы на 90 % за счет слияния многих (но не всех) сервисов распределенной микросервисной архитектуры в монолит, рассматривается здесь: https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90

А.2. НЕДОСТАТКИ МОНОЛИТОВ

У монолитов существует ряд недостатков по сравнению с микросервисами:

- Многие виды функциональности не имеют собственного жизненного цикла, что затрудняет применение методологий Agile.
- Необходимость повторного развертывания всего приложения, чтобы применить какое-либо изменение.
- Большой размер дистрибутива. Высокие требования к ресурсам. Длительное время запуска.
- Необходимость масштабирования как единого приложения.

- Ошибка или нестабильность в любой части монолита может вызвать сбои в ходе эксплуатации.
- Необходимость разработки на одном языке и, как следствие, невозможность использования преимуществ других языков и их фреймворков для специфических требований разных сценариев использования.

А.3. ПРЕИМУЩЕСТВА СЕРВИСОВ

Преимущества сервисов перед монолитами:

1. Быстрая и гибкая разработка, масштабирование требований к продукту/бизнес-функциональности.
2. Модульность и заменяемость.
3. Изоляция сбоев и отказоустойчивость.
4. Более четкие отношения принадлежности и организационная структура.

А.3.1. Быстрая и гибкая разработка, масштабирование требований к продукту и бизнес-функциональности

Проектирование, реализация и развертывание программных продуктов с монолитной архитектурой происходят медленнее, чем в случае сервисов, поскольку для монолитов характерен больший объем кода и сильнее связанные зависимости.

В ходе разработки сервиса можно сосредоточиться на небольшом наборе взаимосвязанных функциональностей и интерфейсе сервиса. Сервисы взаимодействуют через сетевые вызовы в интерфейсах. Иначе говоря, сервисы взаимодействуют через свои определенные API по отраслевым протоколам, таким как HTTP, gRPC и GraphQL. У сервисов существуют очевидные границы в форме их API, тогда как у монолитов их нет. В монолите любой отдельный фрагмент кода обычно имеет многочисленные зависимости, разбросанные по всему коду, и в ходе разработки монолита приходится учитывать всю систему.

С контейнерной инфраструктурой на базе облачных технологий сервис можно разработать и развернуть намного быстрее эквивалентного монолита. Сервис, предоставляющий четко определенный и взаимосвязанный набор возможностей, может интенсивно расходовать ресурсы процессора или памяти, и можно выбрать для него оптимальное оборудование, проводя незатратное повышающее или понижающее масштабирование по мере необходимости. Монолит, предоставляющий разнообразную функциональность, масштабировать подобным образом для оптимизации отдельно взятой функциональности не получится.

Изменения в отдельных сервисах развертываются независимо от других сервисов. По сравнению с монолитом сервис имеет меньший размер пакета, более низкие требования к ресурсам и меньшее время запуска.

А.3.2. Модульность и заменяемость

Независимая природа сервисов делает их модульными и упрощает их замену. Можно реализовать другой сервис с тем же интерфейсом и подставить его на место существующего. В монолите другие разработчики могут изменять код и интерфейсы одновременно с вами, и вам будет сложнее координировать такую разработку по сравнению с сервисами.

Вы можете выбирать технологии, которые лучше подходят для требований сервиса (например, конкретный язык программирования для фронтенда, бэкенда, мобильных приложений или сервисов аналитики).

А.3.3. Изоляция сбоев и отказоустойчивость

В отличие от монолита, архитектура на базе микросервисов не имеет единой точки отказа. Каждый сервис может отслеживаться по отдельности, так что любые отказы можно немедленно сузить до конкретного сервиса. В монолите одна ошибка времени выполнения может вызвать фатальный сбой хоста, влияющий на всю остальную функциональность. Сервис с хорошими практиками отказоустойчивости может адаптироваться к высокой задержке и недоступности других сервисов, от которых он зависит. Такие практики, включая кэширование ответов других сервисов или экспоненциальную задержку с повтором, рассматривались в разделе 3.3. Сервис также может вернуть информированный ответ ошибки вместо сообщения о сбое.

Одни сервисы имеют большую важность, другие — меньшую. Например, какие-то сервисы сильнее влияют на доход или более заметны для пользователей. Наличие отдельных сервисов позволяет классифицировать их по важности и соответствующим образом распределить ресурсы разработки и текущих операций.

А.3.4. Принадлежность и организационная структура

При столь четких границах определить принадлежность сервисов к рабочим группам относительно просто по сравнению с монолитами. Это позволяет концентрировать опыт и знания предметной области; иначе говоря, команда, которой принадлежит конкретный сервис, в ходе его разработки накапливает понимание и практический опыт. Недостаток заключается в том, что команда одного сервиса с меньшей вероятностью разберется в других сервисах и, как следствие, в системе в целом. С другой стороны, монолит побуждает разработчиков понять систему за пределами компонентов, за разработку и обслуживание которых они отвечают. Например, если разработчику потребуются изменения в другом сервисе, он может отправить запрос на реализацию этих изменений соответствующей команде, а не вносить их самостоятельно; это увеличивает время разработки и затраты на коммуникацию. Внесение таких изменений

разработчиком, хорошо знакомым с сервисом, может занять меньше времени и уменьшить риск возникновения ошибок или технического долга.

Природа сервисов с их четко определенными границами также допускает применение разных архитектурных стилей для средств определения API, включая OpenAPI для REST, буферы протоколов для gRPC и SDL (Schema Deinition Language) для GraphQL.

А.4. НЕДОСТАТКИ СЕРВИСОВ

К недостаткам сервисов по сравнению с монолитами можно отнести дублирование компонентов, а также затраты на разработку и обслуживание дополнительных компонентов.

А.4.1. Дублирование компонентов

Каждый сервис должен реализовать внутрисервисные коммуникации и средства безопасности, что, по сути, означает дублирование усилий между сервисами. Надежность системы определяется надежностью самой слабой ее точки, а большое количество сервисов создает большую защищаемую поверхность по сравнению с монолитом.

Разные команды, разрабатывающие дублируемые компоненты, также могут дублировать ошибки и усилия, необходимые для выявления и исправления этих ошибок, что означает неэффективность разработки и обслуживания. Дублирование событий и трата времени также распространяются на пользователей и группу эксплуатации дублируемых сервисов, которые сталкиваются с ошибками, возникающими по причине дубликатов, и тратят лишние усилия на диагностику и общение с разработчиками.

Сервисы не должны совместно использовать базы данных, в противном случае они перестают быть независимыми. Например, изменение в схеме базы данных одного сервиса нарушит работу других сервисов. Отказ от совместного использования баз данных может привести к дублированию данных и увеличению общего объема хранения данных в системе, а также затрат на хранение. Он также может усложнить обеспечение соответствия системы нормам конфиденциальности данных и повысит затраты на него.

А.4.2. Затраты на разработку и обслуживание дополнительных компонентов

Чтобы ориентироваться в разнообразных сервисах организации и понимать их, вам понадобится реестр сервисов и, возможно, дополнительные сервисы для обнаружения сервисов.

Монолитное приложение имеет единый жизненный цикл развертывания. Микросервисной архитектуре приходится управлять разными развертываниями, так что непрерывная интеграция и непрерывное развертывание (CI/CD) становятся необходимостью. К этой же категории относятся такие инфраструктурные элементы, как контейнеры (Docker), реестр контейнеров, оркестрация контейнеров (Kubernetes, Docker Swarm, Mesos), CI-инструменты (такие, как Jenkins) и CD-инструменты, способные поддерживать такие паттерны развертывания, как сине-зеленое развертывание (blue/green deployment), канареечное (canary) и A/B-тестирование.

Когда сервис получает запрос, он может отправить запросы к нижележащим сервисам в процессе обработки этого запроса, который, в свою очередь, может отправить запросы к другим нижележащим сервисам. Структура показана на рис. А.1. Один запрос к домашней странице Netflix расходится по нескольким нижележащим сервисам. Каждый такой запрос добавляет сетевую задержку. Конечная точка сервиса может иметь 1-секундную задержку 99-го процентиля по условиям SLA, но если несколько конечных точек имеют зависимости друг от друга (например, сервис A вызывает сервис B, который вызывает сервис C, и т. д.), задержка для исходного источника запроса может оказаться высокой.

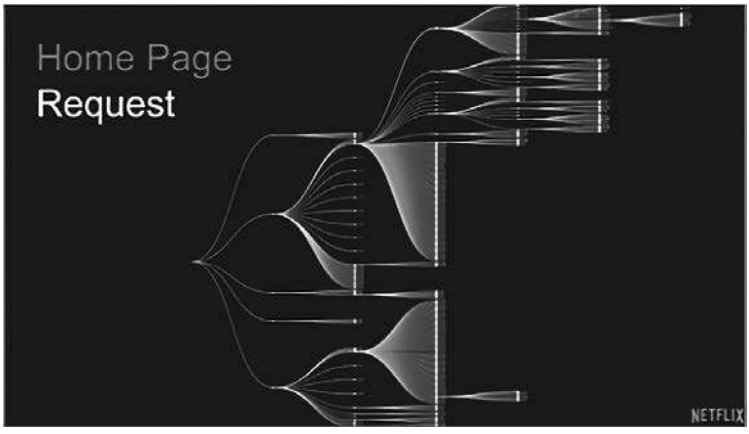

Рис. А.1. Расхождение по нижележащим сервисам запроса к домашней странице Netflix. Изображение взято со страницы https://www.oreilly.com/content/application-caching-at-netlix-the-hidden-microservice/

Кэширование — один из способов решения этой проблемы, но оно создает дополнительную сложность (например, необходимость учитывать политики ограничения срока действия и обновления кэша для предотвращения устаревания данных), а также предполагает затраты на разработку и обслуживание сервиса распределенного кэширования.

Сервис может создать дополнительную сложность, а также потребовать затрат на разработку и обслуживание реализации экспоненциальной задержки

с повтором (см. раздел 3.3.4) на случай сбоев других сервисов, к которым он отправляет запросы.

Другой сложный дополнительный компонент, необходимый архитектуре микросервисов, — распределенная трассировка, используемая для мониторинга и диагностики распределенных систем на базе микросервисов. Jaeger и Zipkin — популярные системы распределенной трассировки.

Установка/обновление библиотеки в монолите требует обновления одного экземпляра этой библиотеки в монолите. В случае сервисов установка/обновление библиотеки, используемой несколькими сервисами, означает ее установку/обновление для всех этих сервисов. Если обновление подразумевает критические изменения, разработчики каждого сервиса должны вручную обновить свои библиотеки и обновить неработоспособный код или конфигурации, обусловленные обратной несовместимостью. Затем они должны развернуть свои обновления с применением средств CI/CD (непрерывной интеграции и непрерывного развертывания) — возможно, последовательно в нескольких средах, прежде чем выполнить итоговое развертывание в рабочей среде. Они должны вести мониторинг этих развертываний. В процессе разработки и развертывания они также должны диагностировать любые непредвиденные проблемы. Задача может сводиться к копированию сообщений об ошибках для поиска решений в Google или во внутреннем чат-приложении компании (например, Stack или Microsoft Teams). Если развертывание завершится неудачей, разработчик должен провести диагностику, а потом повторить развертывание и посмотреть, завершится ли оно успехом или неудачей. При этом иногда приходится иметь дело со сложными сценариями (например, постоянными сбоями на некотором хосте). Все это создает существенную дополнительную нагрузку на разработчика. Более того, дублирование логики и библиотек также может привести к заметному увеличению затрат памяти.

А.4.3. Распределенные транзакции

Сервисы используют отдельные базы данных, поэтому для обеспечения согласованности этих баз данных могут потребоваться распределенные транзакции — в отличие от монолита с одной реляционной базой данных, которая отправляет транзакции к этой базе данных. Необходимость реализации распределенных транзакций — еще один источник затрат, сложности, задержки, возможных ошибок и сбоев. Распределенные транзакции рассматривались в главе 5.

А.4.4. Ссылочная целостность

Ссылочной целостностью называется точность и консистентность данных, между которыми существует связь. Если значение одного атрибута в связи ссылается на значение другого атрибута, то значение, на которое указывает ссылка, должно существовать.

Ссылочная целостность в единой базе данных монолита легко реализуется с использованием внешних ключей. Значения в столбце внешнего ключа должны либо присутствовать в первичном ключе, на который ссылается внешний ключ, либо быть равными `null` (https://www.interfacett.com/blogs/referential-integrity-options-cascade-set-null-and-set-default). Ссылочная целостность усложняется, если базы данных распределяются между сервисами. Для ссылочной целостности в распределенной системе запрос записи, в котором задействованы несколько сервисов, должен завершиться успехом или неудачей/отменой/откатом в каждом сервисе. Процесс записи должен включать такие шаги, как повторные попытки и/или откаты/компенсирующие транзакции. Более подробно распределенные транзакции рассматривались в главе 5. Для проверки ссылочной целостности также может понадобиться периодический аудит всех сервисов.

А.4.5. Координация разработки функциональности и развертывания для нескольких сервисов

Если новая функциональность охватывает несколько сервисов, разработку и развертывание приходится координировать между ними. Например, один сервис API может зависеть от других. Так, команде разработки сервиса REST-совместимого API Rust Rocket (https://rocket.rs/) может понадобиться разработать новые конечные точки API, которые будут использоваться сервисом React UI, разрабатываемым другой командой UI. Рассмотрим этот пример более подробно.

В теории разработка функциональности может проходить параллельно в обоих сервисах. Команда API должна лишь предоставить спецификацию новых конечных точек API. Команда UI может разработать новые компоненты React и сопутствующий серверный код Express или node.js. Так как команда API еще не предоставила тестовую среду, которая возвращает реальные данные, серверный код использует макеты или заглушки ответов от новых конечных точек API. Этот метод также эффективен для создания модульных тестов в коде UI, включая spy-тесты (за дополнительной информацией обращайтесь по адресу https://jestjs.io/docs/mock-function-api).

Команды также могут использовать флаги функциональности для избирательного предоставления незавершенной функциональности в средах разработки и промежуточных средах, скрывая ее в рабочей среде. Это позволяет другим разработчикам и ключевым участникам, заинтересованным в новой функциональности, просматривать и обсуждать незавершенную работу.

На практике ситуация может оказаться намного более сложной. Понять все нюансы нового набора конечных точек API может быть сложно даже разработчикам и UX-дизайнерам со значительным опытом работы с этим API. Нестандартные проблемы могут обнаруживать как разработчики API, так и разработчики UI в ходе проектирования соответствующих сервисов; в API придется вносить

изменения, и обе команды будут вынуждены обсуждать решение, и, возможно, часть работы будет выполнена впустую:

- Модель данных может не подходить для UX. Например, при разработке функциональности контроля версий для шаблонов системы оповещений (см. раздел 9.5) UX-дизайнер может спроектировать интерфейс контроля версий так, чтобы в нем учитывались отдельные шаблоны. Но шаблон может включать подкомпоненты, которые версионируются отдельно. Эта путаница может быть выявлена, только когда и разработчики UI, и разработчики API уже проделают часть работы.
- В ходе разработки команда API может обнаружить, что новые конечные точки API требуют неэффективных запросов к базам данных, например слишком больших запросов SELECT или операций JOIN между большими таблицами.
- Для REST или RPC API (не GraphQL) пользователю может потребоваться отправить несколько запросов API, а затем выполнить сложные операции постобработки с ответами, прежде чем данные будут возвращены источнику запроса или выведены в UI. Или предоставленный API может загружать намного больше данных, чем требует UI, что увеличивает задержку. Если API разрабатываются внутренними силами, команда UI может потребовать перепроектирования и переработки API, чтобы сделать запросы менее сложными и более эффективными.

А.4.6. Интерфейсы

Сервисы могут быть написаны на разных языках и взаимодействовать друг с другом по текстовому или двоичному протоколу. В случае текстовых протоколов (таких, как JSON или XML) необходимо преобразовывать эти строки в объекты и обратно. Также приходится писать дополнительный код, необходимый для проверки данных и обработки ошибок и исключений для отсутствующих полей. Для корректного сокращения функциональности сервису может потребоваться обрабатывать объекты с отсутствующими полями. Чтобы обработать возможный возврат таких данных от зависимых сервисов, необходимо реализовать резервный вариант, например кэширование данных от зависимых сервисов и возврат старых данных или, как еще один вариант, возврат данных с отсутствующими полями. Это может привести к тому, что реализация будет отличаться от документации.

А.5. ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ

В этом приложении использован материал из книги Касуна Индрасири (Kasun Indrasiri) и Прабата Сиривардены (Prabath Siriwardena) «Microservices for the Enterprise: Designing, Developing, and Deploying» (2018, Apress).

Б Авторизация OAuth 2.0 и аутентификация OpenID Connect[1]

Б.1. АВТОРИЗАЦИЯ И АУТЕНТИФИКАЦИЯ

Авторизацией называется процесс предоставления пользователю (человеку или системе) разрешений на обращение к конкретному ресурсу или функции. *Аутентификация* представляет собой проверку идентичности пользователя. *OAuth 2.0* — популярный алгоритм авторизации. (Протокол OAuth 1.0 был опубликован в апреле 2010 года, а протокол OAuth 2.0 — в октябре 2012 года) *OpenID Connect* представляет собой расширение OAuth 2.0 для аутентификации. Аутентификация и авторизация/управление доступом — типичные требования к безопасности сервиса. OAuth 2.0 и OpenID Connect можно кратко обсудить на собеседовании в контексте авторизации и аутентификации.

Одно из заблуждений, часто встречающихся в Сети, — идея «входа с использованием OAuth2». Такие ресурсы путают концепции авторизации и аутентификации. В этом приложении содержится краткий вводный курс авторизации с помощью OAuth2 и аутентификации с помощью OpenID Connect, а также поясняются различия между авторизацией и аутентификацией.

Б.2. ВВЕДЕНИЕ: ПРОСТОЙ ВХОД, АУТЕНТИФИКАЦИЯ НА БАЗЕ COOKIE

Простейшая разновидность аутентификации обычно называется *простым входом*, *базовой аутентификацией* или *аутентификацией на базе форм*. При

[1] В этом разделе использован материал из видео OAuth 2.0 and OpenID Connect (in plain English), http://oauthacademy.com/talk — отличной вводной лекции Нейта Барбеттини (Nate Barbettini) и https://auth0.com/docs. Дополнительную информацию также см. на https://oauth.net/2/.

простом входе пользователь вводит пару (*идентификатор*, *пароль*). Часто используются варианты (*имя_пользователя*, *пароль*) и (*электронная_почта*, *пароль*).

Когда пользователь отправляет свое имя пользователя и пароль, бэкенд проверяет, что пароль для указанного имени пользователя введен верно. В целях безопасности пароли необходимо хешировать с использованием соли. После верификации бэкенд создает сеанс для этого пользователя и задает значение cookie, которое будет храниться как в памяти сервиса, так и в браузере пользователя. UI задает cookie в браузере пользователя: например, `Set-Cookie: sessionid=f00b4r; Max-Age: 86400;`. Здесь значение cookie содержит идентификатор сеанса. Дальнейшие запросы от браузера используют идентификатор сеанса для аутентификации, чтобы пользователю не приходилось снова вводить имя пользователя и пароль. Каждый раз при отправке запроса к бэкенду браузер отправляет на бэкенд идентификатор сеанса, а бэкенд сравнивает отправленный идентификатор сеанса со своей копией для проверки идентичности пользователя.

Этот процесс называется *аутентификацией на базе cookie*. Сеанс имеет конечную продолжительность по времени, по истечении которого пользователь должен заново ввести имя и пароль. У сеансов существуют два типа тайм-аутов: абсолютный и по отсутствию активности. *Абсолютный тайм-аут* завершает сеанс после истечения заданного периода, *тайм-аут по отсутствию активности* — после заданного периода, во время которого пользователь не взаимодействует с приложением.

Б.3. ЕДИНЫЙ ВХОД

Единый вход (SSO, Single Sign-On) дает возможность пользователю входить в разные системы под одной главной учетной записью (например, учетной записью Active Directory). SSO обычно реализуется по протоколу, который называется SAML (Security Assertion Markup Language). Появление мобильных приложений в конце 2000-х имело ряд следствий:

- Cookie не подходят для устройств, поэтому для сеансов с долгим сроком жизни требовался новый механизм, сохраняющий активный статус пользователя в мобильном приложении даже после закрытия приложения.
- Новый сценарий использования назывался *делегированной авторизацией*. Владелец набора ресурсов может делегировать доступ к некоторым, но не ко всем этим ресурсам заданному клиенту. Например, можно предоставить некоторому приложению разрешение на просмотр некоторых видов информации пользователя Facebook — скажем, его открытого профиля и дня рождения, но не постов на стене.

Б.4. НЕДОСТАТКИ ПРОСТОГО ВХОДА

Недостатки простого входа — сложность, трудности с обслуживанием и отсутствие частичной авторизации.

Б.4.1. Сложность и трудности с обслуживанием

Основные компоненты простого входа (или аутентификации на базе сеансов вообще) реализуются разработчиком:

- Конечная точка и логика входа, включая добавление соли и хеширование.
- Таблица базы данных с именами пользователей и засоленными и хешированными паролями.
- Создание и сброс паролей, включая операции 2FA (такие, как сообщения о сбросе пароля).

Это означает, что разработчик приложения отвечает за соблюдение лучших практик безопасности. В OAuth 2.0 и OpenID Connect паролями управляет отдельный сервис. (Это относится ко всем протоколам на базе токенов. OAuth 2.0 и OpenID Connect — протоколы на базе токенов.) Разработчик приложения может использовать сторонний сервис с хорошими практиками безопасности, чтобы снизить риск взлома паролей.

Cookie требуют сервиса для хранения состояния. Каждому пользователю, выполнившему вход, необходим сервер, который создаст для него сеанс. При миллионах сеансов затраты памяти могут оказаться слишком высокими. У протоколов на базе токенов нет лишних затрат памяти.

Разработчик также несет ответственность за соответствие приложения нормам конфиденциальности пользователя, таким как GDPR (General Data Protection Regulation), CCPA (California Consumer Privacy Act) и HIPAA (Health Insurance Portability and Accountability Act).

Б.4.2. Отсутствие частичной авторизации

У простого входа не существует концепции частичных разрешений управления доступом. Возможно, вы захотите предоставить другой стороне частичный доступ к чьей-то учетной записи для конкретных целей. Предоставление полного доступа создает риск для безопасности. Например, пользователь хочет разрешить приложению по управлению финансами (такому, как Mint) просматривать баланс его банковского счета, но не предоставлять других разрешений (на перевод денег и т. д.). Если банковское приложение использует только простой вход, такой частичный доступ невозможен. Пользователь должен передать Mint имя пользователя и пароль учетной записи своего банковского приложения, но это

будет означать, что он предоставит Mint полный доступ, а не только доступ на просмотр баланса.

Другой пример — технология Yelp до разработки OAuth. Как показано на рис. Б.1, в конце регистрации пользователя Yelp запрашивает его учетную запись Gmail, чтобы иметь возможность отправить реферальную ссылку или ссылку для приглашения в список контактов. Пользователь должен предоставить Yelp полный доступ к учетной записи Gmail только для того, чтобы отправить одно сообщение каждому из своих контактов.

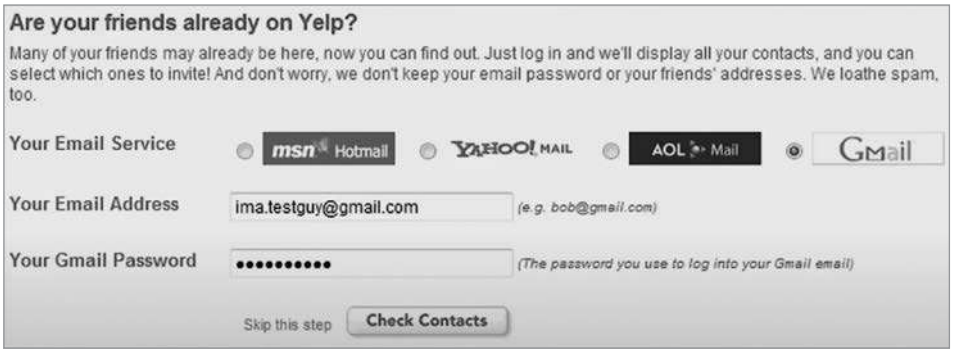

Рис. Б.1. Снимок экрана браузерного приложения Yelp до появления OAuth, демонстрирующий отсутствие частичной авторизации при простом входе. Пользователь должен ввести адрес своей электронной почты и пароль, предоставляя Yelp полный доступ к своей учетной записи, хотя Yelp всего лишь отправит одно сообщение каждому из его контактов. Изображение взято со страницы http://oauthacademy.com/talk

Технология OAuth 2.0 получила широкое распространение, так что большинство приложений такие практики уже не применяет. Заметным исключением является банковская отрасль. По состоянию на 2022 год большинство банков еще не приняло технологию OAuth.

Б.5. ПОСЛЕДОВАТЕЛЬНОСТЬ ДЕЙСТВИЙ OAUTH 2.0

В этом разделе рассматривается последовательность действий OAuth 2.0 и объясняется, как такие приложения, как Google, используют OAuth 2.0, чтобы пользователь мог разрешить приложениям наподобие Yelp обращаться к ресурсам, принадлежащим пользователю Google (например, отправлять сообщения электронной почты контактам пользователя в Google).

На рис. Б.2 представлены основные этапы взаимодействия между Yelp и Google в OAuth. В этой главе мы будем придерживаться этой схемы.

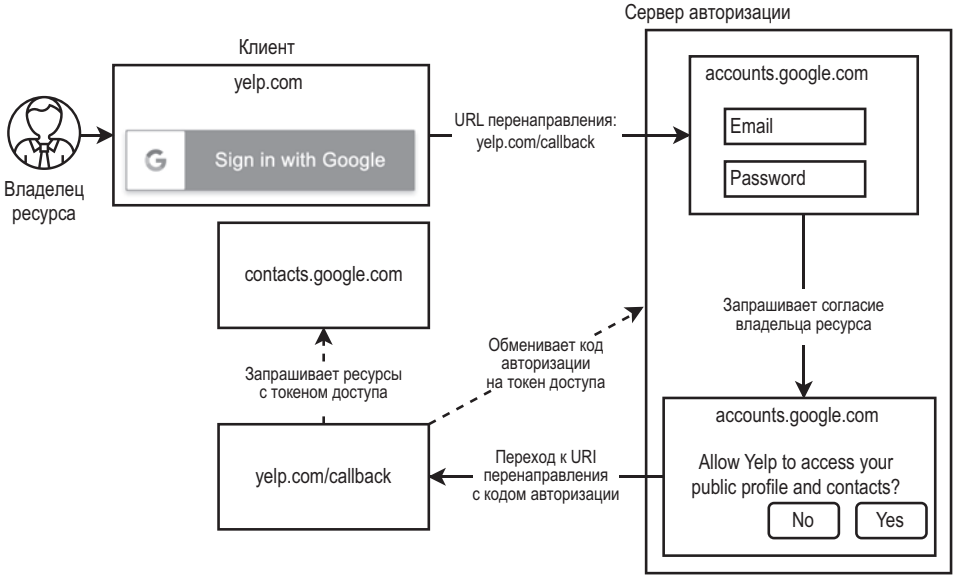

Рис. Б.2. Последовательность действий OAuth 2.0, подробно рассматриваемая в этом разделе. Коммуникации основного канала представлены сплошными линиями, а коммуникации закрытого канала — пунктирными

Б.5.1. Терминология OAuth 2.0

- *Владелец ресурса* — пользователь, который является владельцем данных или контролирует операции, которые запрашивает приложение. Например, если вы храните список контактов в учетной записи Google, вы являетесь владельцем ресурса для этих данных. Вы можете предоставить приложению разрешение на обращение к этим данным. В этом разделе владелец ресурса будет для краткости называться «пользователем».
- *Клиент* — приложение, запрашивающее ресурсы.
- *Сервис авторизации* — система, используемая пользователем для авторизации разрешения (например, accounts.google.com).
- *Сервер ресурса* — API системы, которая хранит данные, нужные клиенту (например, Google Contacts API). В зависимости от системы сервер авторизации и сервер ресурса могут быть как одной, так и разными системами.
- *Предоставление авторизации* — подтверждение согласия пользователя на предоставление разрешений, необходимых для обращения к ресурсам.
- *URI перенаправления*, также называемый *URI обратного вызова*, — URI, по которому сервер авторизации возвращает управление клиенту.

- *Токен доступа* — ключ, используемый клиентом для получения доступа к авторизованному ресурсу.
- *Область действия* — сервер авторизации хранит список поддерживаемых областей действия (например, чтение списка контактов пользователя в Google, чтение электронной почты или удаление электронной почты). Клиент может запросить конкретный набор областей действия в зависимости от требуемых ресурсов.

Б.5.2. Исходная настройка клиента

Приложение (такое, как Mint или Yelp) должно провести единоразовую настройку сервера авторизации (например, Google), чтобы стать клиентом и дать пользователю возможность использовать OAuth. Когда Mint отправляет Google запрос на создание клиента, Google предоставляет:

- Идентификатор клиента — как правило, длинный уникальный строковый идентификатор. Передается с исходным запросом по основному каналу.
- Секрет клиента, используемый при обмене токенами.

1. Получение авторизации у пользователя

Поток операций начинается с владельца ресурса (Google) в клиентском приложении (Yelp). Yelp выводит кнопку, при помощи которой пользователь сможет предоставить доступ к определенным данным своей учетной записи Google. Нажатие кнопки включает пользователя в поток операций OAuth — последовательность действий, в результате которой приложение получит авторизацию и сможет обращаться только к запрашиваемой информации.

Когда пользователь нажимает кнопку, браузер перенаправляется на сервер авторизации (например, домен Google, которым может быть accounts.google.com, или сервис авторизации Facebook или Okta). Здесь пользователю предлагается выполнить вход (то есть ввести свой адрес электронной почты и пароль, после чего щелкнуть по кнопке Login (Вход)). Пользователь видит в адресной строке своего браузера, что он находится в домене Google. Тем самым повышается уровень безопасности, так как пользователь предоставляет свой адрес электронной почты и пароль Google, а не приложению, такому как Mint или Yelp.

При этом перенаправлении клиент передает конфигурационную информацию серверу авторизации в запросе с URL вида https://accounts.google.com/o/oauth2/v2/auth?client_id=yelp&redirect_uri=https%3A%2F%2Foidcdebugger.com%2Fdebug&scope=openid&response_type=code&response_mode=query&state=foobar&nonce=uwtukpm946m. Параметры запроса:

- *client_id* — идентифицирует клиента для сервера авторизации, например сообщает Google, что клиентом является Yelp.

- *redirect_uri* (также называемый *URI обратного вызова*) — URI перенаправления.
- *scope* — список запрашиваемых областей действия.
- *response_type* — тип предоставления авторизации, нужный клиенту. Существуют несколько типов, которые будут описаны ниже. Пока будем предполагать самый распространенный тип, называемый *предоставлением кода авторизации*, — запрос к серверу авторизации для получения кода.
- *state* — состояние передается от клиента обратному вызову. Как упоминается ниже, на шаге 4, тем самым предотвращаются атаки межсайтовой подделки запросов (cross-site request forgery, CSRF).
- *nonce* — сокращение от number used once, то есть «одноразовое число»: случайное значение, предоставляемое сервером и используемое для уникальной пометки запроса, чтобы предотвратить атаки воспроизведения (эта тема в книге не рассматривается).

2. Пользователь дает согласие на области действия клиента

После выполнения входа сервер авторизации предлагает пользователю дать согласие на запрашиваемый клиентом список областей действия. В нашем примере Google выводит сообщение со списком ресурсов, запрашиваемых другим приложением (например, общедоступный профиль и список контактов), и предлагает подтвердить, что пользователь согласен предоставить эти ресурсы приложению. Тем самым гарантируется, что пользователь не будет введен в заблуждение и случайно не предоставит доступ к какому-либо ресурсу, к которому предоставлять доступ не собирался.

Независимо от того, ответит ли пользователь согласием или отказом, браузер перенаправляется обратно по URI обратного вызова приложения с другими параметрами запроса в зависимости от решения пользователя. Если пользователь ответит отказом, приложение не получает доступа. URI перенаправления может иметь вид https://yelp.com/callback?error=access_denied&error_description=The user did not consent. Если же пользователь согласится, приложение может запросить предоставленные ресурсы пользователя через Google API, например API Google Contacts. Сервер авторизации передает управление по URI перенаправления с кодом авторизации. URI перенаправления может иметь вид https://yelp.com/callback?code=3mPDQbnIOyseerTTKPV&state=foobar, где параметр запроса *code* содержит код авторизации.

3. Запрос токена доступа

Клиент отправляет запрос POST серверу авторизации, чтобы обменять код авторизации на токен доступа, включающий секретный ключ клиента (известный только клиенту и серверу авторизации). Пример:

```
POST www.googleapis.com/oauth2/v4/token
Content-Type: application/x-www-form-urlencoded

code=3mPDQbnIOyseerTTKPV&client_id=yelp&client_secret=secret123&grant_
type=authorization_code
```

Сервер авторизации проверяет код и отвечает токеном доступа и состоянием, полученным от клиента.

4. Запрос ресурсов

Для предотвращения CSRF-атак клиент проверяет, что состояние, отправленное серверу, идентично состоянию в ответе. Затем клиент использует токен доступа для запроса авторизованных ресурсов от сервера ресурсов. Токен доступа позволяет клиенту обратиться только к запрашиваемой области действия (например, доступ только для чтения к контактам Google пользователя). Запросы других ресурсов, находящихся за пределами области действия или в других областях, будут отклоняться (например, удаление контактов или обращение к истории пользователя):

```
ET api.google.com/some/endpoint
Authorization: Bearer h9pyFgK62w1QZDox0d0WZg
```

Б.5.3. Закрытый канал и основной канал

Почему мы получаем код авторизации, а затем обмениваем его на токен доступа? Почему нельзя просто использовать код авторизации или получить токен доступа немедленно? Чтобы понять это, необходимо ознакомиться с концепциями закрытого канала и основного канала — понятиями из области сетевой безопасности.

Коммуникации по основному каналу происходят между двумя и более сторонами, наблюдаемыми в протоколе. *Коммуникации по закрытому каналу* ненаблюдаемы по крайней мере одной стороной в протоколе. Это делает коммуникации по закрытому каналу более безопасными, чем по основному.

Примером закрытого, или высокозащищенного, канала служит запрос HTTP с шифрованием SSL от сервера клиента к серверу Google API. Примером основного канала служит браузер пользователя. Браузер защищен, но в нем есть лазейки или места, в которых возможна утечка данных. Если в веб-приложении хранится секретный пароль или ключ и он включается в HTML или JavaScript, этот секрет становится видимым каждому, кто просмотрит исходный код страницы. Взломщик также может открыть сетевую консоль или инструменты разработчика Chrome, чтобы просмотреть или изменить JavaScript. Браузер считается основным каналом, потому что ему нельзя полностью доверять, но вы полностью доверяете коду, выполняемому на серверах бэкенда.

Рассмотрим ситуацию, в которой клиент действует через сервер авторизации по основному каналу. Полностраничные перенаправления, исходящие запросы,

перенаправления к серверу авторизации и содержимое запроса к серверу авторизации передаются через браузер. Код авторизации также передается через браузер (то есть по основному каналу). Если код авторизации будет перехвачен (например, вредоносной панелью инструментов или механизмом, способным сохранять запросы браузера), взломщик не сможет получить код доступа, потому что обмен токена происходит в закрытом канале.

Обмен токена происходит между бэкендом и каналом авторизации, а не браузером. Бэкенд также включает в обмен свой секретный ключ, неизвестный взломщику. Если бы передача секретного ключа происходила через браузер, взломщик смог бы похитить его, поэтому передача происходит через закрытый канал.

Процесс OAuth 2.0 спроектирован с учетом лучших характеристик основного и закрытого канала, чтобы обеспечить высокую безопасность. Основной канал используется для взаимодействия с пользователем. Браузер, предназначенный для прямого взаимодействия с пользователем и отображения экранов, выводит для пользователя экран входа и экран согласия. Мы не можем полностью доверять браузеру с использованием секретных ключей, поэтому последний шаг процесса (то есть обмен) выполняется в обратном канале — системе, которой вы доверяете.

Сервер авторизации также может выдать токен обновления, чтобы клиент мог получить новый токен доступа в случае истечения срока действия старого без взаимодействия с пользователем. Эта тема в книге не рассматривается.

Б.6. ДРУГИЕ ПРОЦЕССЫ OAUTH 2.0

Мы описали процесс использования кода авторизации, в котором задействован как закрытый, так и основной канал. Другие варианты — неявная передача (только основной канал), передача идентификационных данных пароля владельца ресурса (только обратный канал) и передача идентификационных данных клиента (только обратный канал).

Неявная передача — единственный способ использования OAuth 2.0 при отсутствии бэкенда в приложении. На рис. Б.3 представлен пример неявной передачи. Все коммуникации выполняются только в основном канале. Сервер авторизации возвращает код доступа напрямую, без кода авторизации и без шага обмена.

Неявная передача создает некоторый риск в отношении безопасности, так как токен доступа становится доступным из браузера.

Передача идентификационных данных пароля владельца ресурса и идентификационных данных клиента используется в старых приложениях, в новых приложениях она не рекомендуется. Сервер бэкенда использует свои идентификационные данные для запроса токена доступа у сервера авторизации. Передача идентификационных данных клиента иногда используется в коммуникациях «машина — машина» или между сервисами.

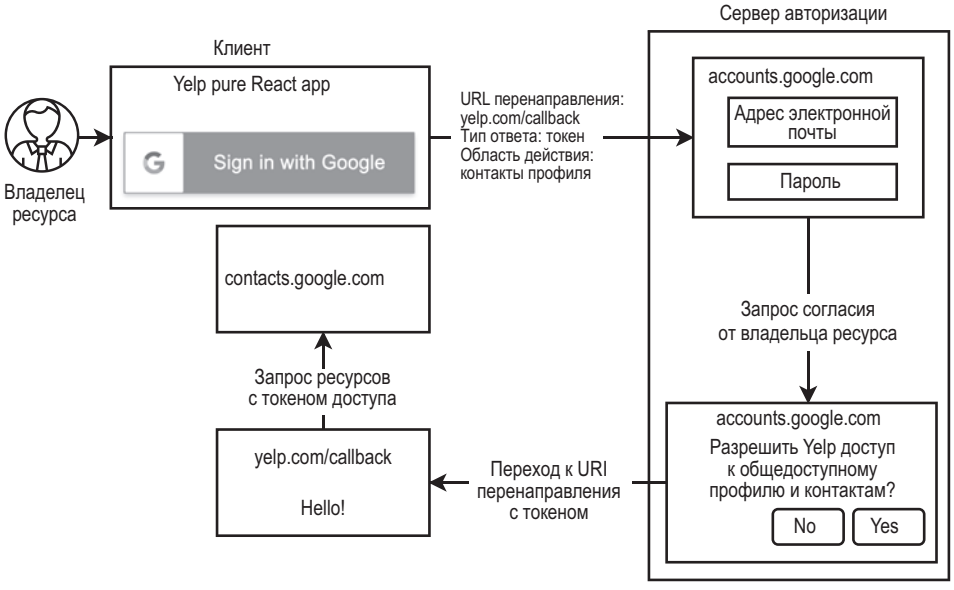

Рис. Б.3. Схема неявной передачи OAuth2. Все коммуникации выполняются только в основном канале. Обратите внимание: запрос к серверу авторизации имеет тип ответа «токен» вместо «код»

Б.7. АУТЕНТИФИКАЦИЯ OPENID CONNECT

Кнопка Login with Facebook (Войти с аккаунтом Facebook) появилась в 2009 году; за ней последовала кнопка Login with Google и аналогичные кнопки многих других компаний: Twitter, Microsoft, LinkedIn и т. д. Они позволяют выполнить вход на сайт с существующими идентификационными данными Facebook, Google или других социальных сетей. Такие кнопки получили повсеместное распространение. Они хорошо подходили для своей задачи и создавались с использованием OAuth 2.0, хотя протокол OAuth 2.0 не предназначался для аутентификации. По сути, OAuth 2.0 использовался нецелевым образом — для делегированной авторизации.

Однако использование OAuth для аутентификации считается плохой практикой, поскольку в OAuth отсутствует возможность получения информации пользователя. Когда вы входите в приложение с OAuth 2.0, оно не сможет узнать, кто только что вошел, или получить информацию о вас, например адрес электронной почты и имя. Протокол OAuth 2.0 проектировался для областей действия разрешений. Он всего лишь проверяет, что токен доступа распространяется на конкретный набор ресурсов. Он не проверяет, кто вы.

Когда разные компании проектируют кнопки входа через социальные сети, используя OAuth во внутренней реализации, им всем приходится применять

особые трюки поверх OAuth, чтобы клиенты могли получать информацию пользователя. Изучая эти реализации, помните, что они отличаются друг от друга и несовместимы между собой.

Чтобы решить проблему нехватки стандартизации, была создана технология OpenID как стандарт использования OAuth 2.0 для аутентификации. OpenID Connect представляет собой тонкий слой поверх OAuth 2.0, позволяющий применять ее для аутентификации. OpenID Connect добавляет к OAuth 2.0 следующее:

- *Идентификационный токен* — представляет идентификатор пользователя и содержит некоторую информацию о пользователе. Этот токен возвращается сервером авторизации во время обмена.
- *Конечная точка информации о пользователе* — если клиенту нужно больше информации, чем содержится в идентификационном токене, возвращаемом сервером авторизации, клиент может запросить ее из конечной точки информации о пользователе.
- *Стандартный набор областей действия*.

Таким образом, технические различия между OAuth 2.0 и OpenID Connect сводятся к тому, что OpenID Connect возвращает как код доступа, так и идентификационный токен, а OpenID Connect предоставляет конечную точку информации о пользователе. Клиент может запросить у сервера авторизации область действия OpenID в дополнение к нужным областям видимости OAuth 2.0 и получить код доступа и идентификационный токен.

В табл. Б.1 приведена сводка сценариев использования OAuth 2.0 (авторизация) и OpenID Connect (аутентификация).

Таблица Б.1. Сценарии использования OAuth 2.0 (авторизация) и OpenID Connect (аутентификация)

OAuth2 (авторизация)	OpenID Connect (аутентификация)
Предоставляет доступ к вашему API	Выполняет вход пользователя в систему
Предоставляет доступ к пользовательским данным в других системах	Предоставляет доступ к учетным записям пользователя в других системах

Идентификационный токен состоит из трех частей:

- *Заголовок* — содержит несколько полей (например, алгоритм, используемый для кодирования цифровой подписи).
- *Заявки* — тело/полезная информация идентификационного токена. Клиент декодирует заявки для получения информации о пользователе.

- *Цифровая подпись* — клиент может использовать ее, чтобы проверить, что идентификационный токен не изменился. Другими словам, подпись может независимо проверяться клиентским приложением без контакта с сервером авторизации.

Клиент также может использовать токен доступа для запроса к конечной точке информации о пользователе на сервере авторизации, чтобы получить такую информацию (например, изображение в профиле пользователя). В табл. Б.2 описано, какой тип предоставления должен использоваться в каждом сценарии.

Таблица Б.2. Тип предоставления, используемый в каждом конкретном случае

Веб-приложение с серверным бэкендом	Передача кода авторизации
Нативное мобильное приложение	Передача кода авторизации с PKCE (Proof Key for Code Exchange) (в книге не рассматривается)
Одностраничные приложения (SPA) JavaScript с API бэкенда	Неявная передача
Микросервисы и API	Передача идентификационных данных клиента

B Модель C4

Модель C4 (https://c4model.com/) — метод разложения системы на разные уровни абстракции, разработанный Саймоном Брауном (Simon Brown) и основанный на диаграммах системной архитектуры. В этом приложении приведено краткое введение в модель C4. На указанном сайте представлено хорошее введение и глубокий анализ модели C4, так что мы ограничимся ее кратким описанием; за дополнительной информацией читателям стоит обращаться на веб-сайт. Модель C4 определяет четыре уровня абстракции.

Контекстная диаграмма представляет систему одним блоком, вокруг которого размещаются пользователи и другие системы, с которыми она взаимодействует. На рис. B.1 приведен пример контекстной диаграммы новой системы интернет-банка, проектируемой на базе существующей системы. Ее пользователями будут клиенты банка, пользующиеся интернет-банком через UI-приложения, которые мы будем разрабатывать для них. Наша система интернет-банка также использует готовую систему электронной почты. На рис. B.1 пользователи и системы изображены в виде прямоугольников и соединяются стрелками, представляющими запросы между ними.

Контейнерная диаграмма определяется на сайте *c4model.com* как «отдельно запускаемая/развертываемая единица, которая выполняет код или хранит данные». Мы также понимаем контейнеры как сервисы, образующие систему. На рис. B.2 представлен пример контейнерной диаграммы. На ней система интернет-банка, представленная одним блоком на рис. B.1, поделена на составляющие. Пользователь может загрузить одностраничное (браузерное) приложение из сервиса веб-приложения, а затем продолжить отправлять запросы к одностраничному приложению. Мобильный пользователь может загрузить мобильное приложение из интернет-магазина и отправлять все запросы через это приложение.

Браузерные и мобильные приложения отправляют запросы к приложению/сервису API (бэкенда). Сервис бэкенда отправляет запросы к базе данных SQL Oracle, мейнфреймовой банковской системе и системе электронной почты.

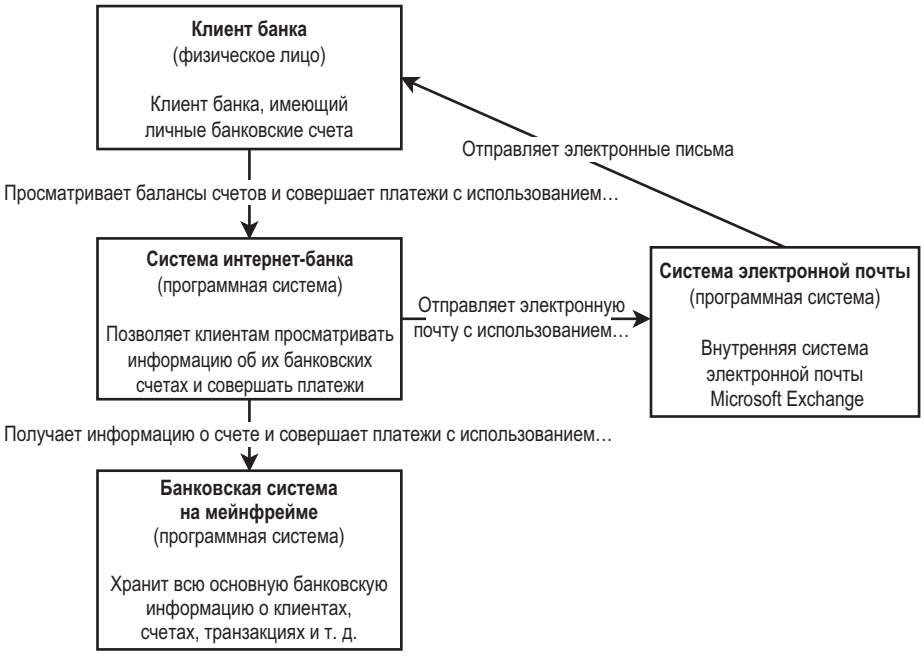

Рис. В.1. Контекстная диаграмма. Изображение с сайта https://c4model.com/, публикуется на условиях лицензии https://creativecommons.org/licenses/by/4.0/. В данном случае проектируется система интернет-банка. Ее пользователями являются клиенты банка — физические лица, пользующиеся интернет-банком через UI-приложения. Система интернет-банка отправляет запросы к унаследованной системе, работающей на мейнфрейме. Также она использует существующую систему электронной почты для рассылки сообщений пользователям. Многие другие общие сервисы, которые она может использовать, еще недоступны и могут обсуждаться в ходе проектирования

Компонентная диаграмма представляет набор классов, находящихся за интерфейсом, для реализации функциональности. Компоненты не являются независимо развертываемыми блоками. На рис. 6.3 представлен пример компонентной диаграммы приложения/сервиса API (бэкенда) с рис. 6.2. На нем показаны интерфейсы и классы и их запросы к другим сервисам.

Браузерные и мобильные приложения отправляют запросы к бэкенду, которые маршрутизируются к соответствующим интерфейсам.

Контроллер входа получает запросы на вход. Контроллер сброса паролей получает запросы на сброс паролей. Компонент безопасности содержит функции для обработки функциональности, связанной с безопасностью, от контроллера входа и контроллера сброса пароля. Он сохраняет данные в базе данных SQL Oracle.

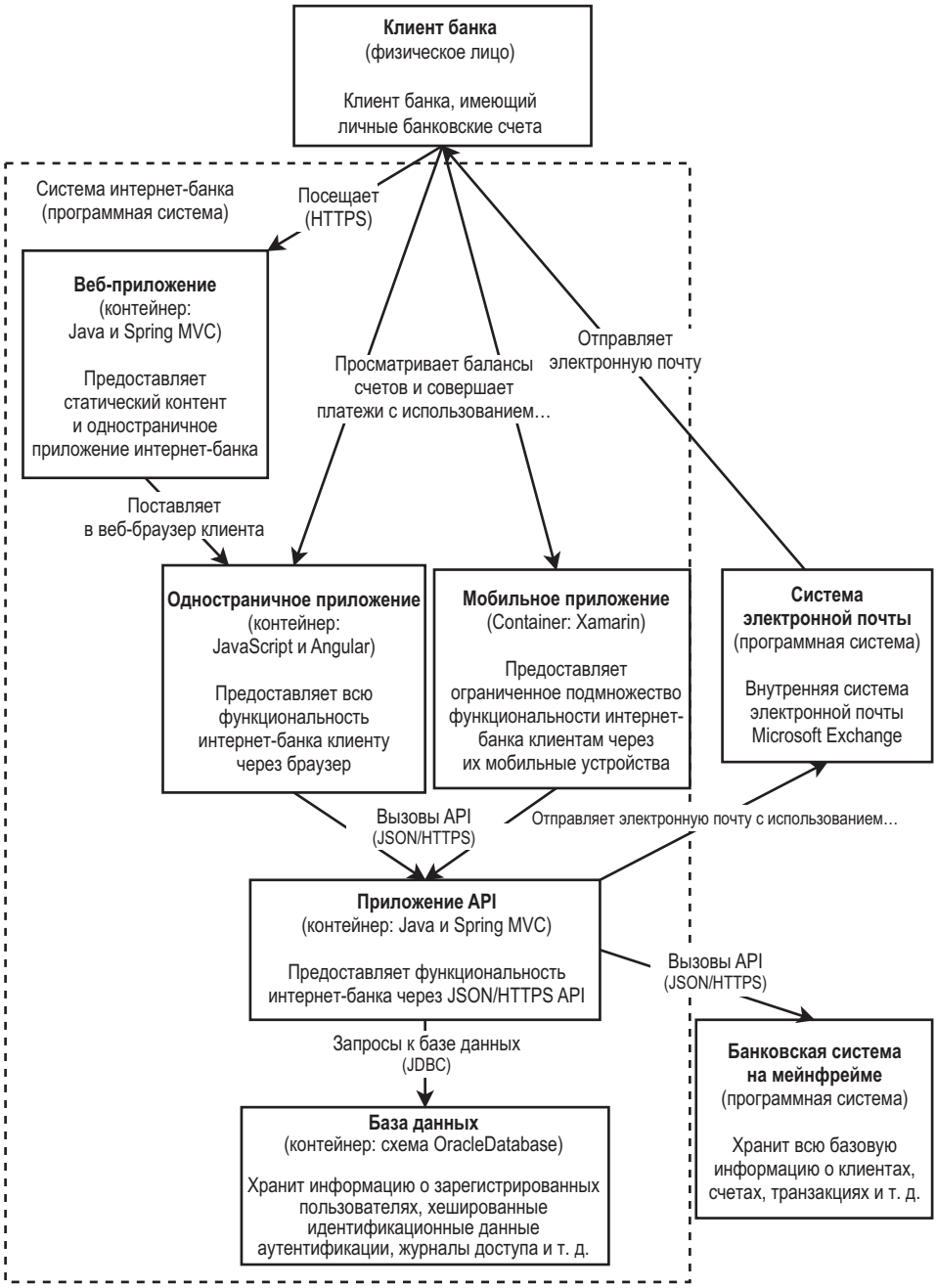

Рис. В.2. Контейнерная диаграмма. Изображение с сайта https://c4model.com/, публикуется на условиях лицензии https://creativecommons.org/licenses/by/4.0/

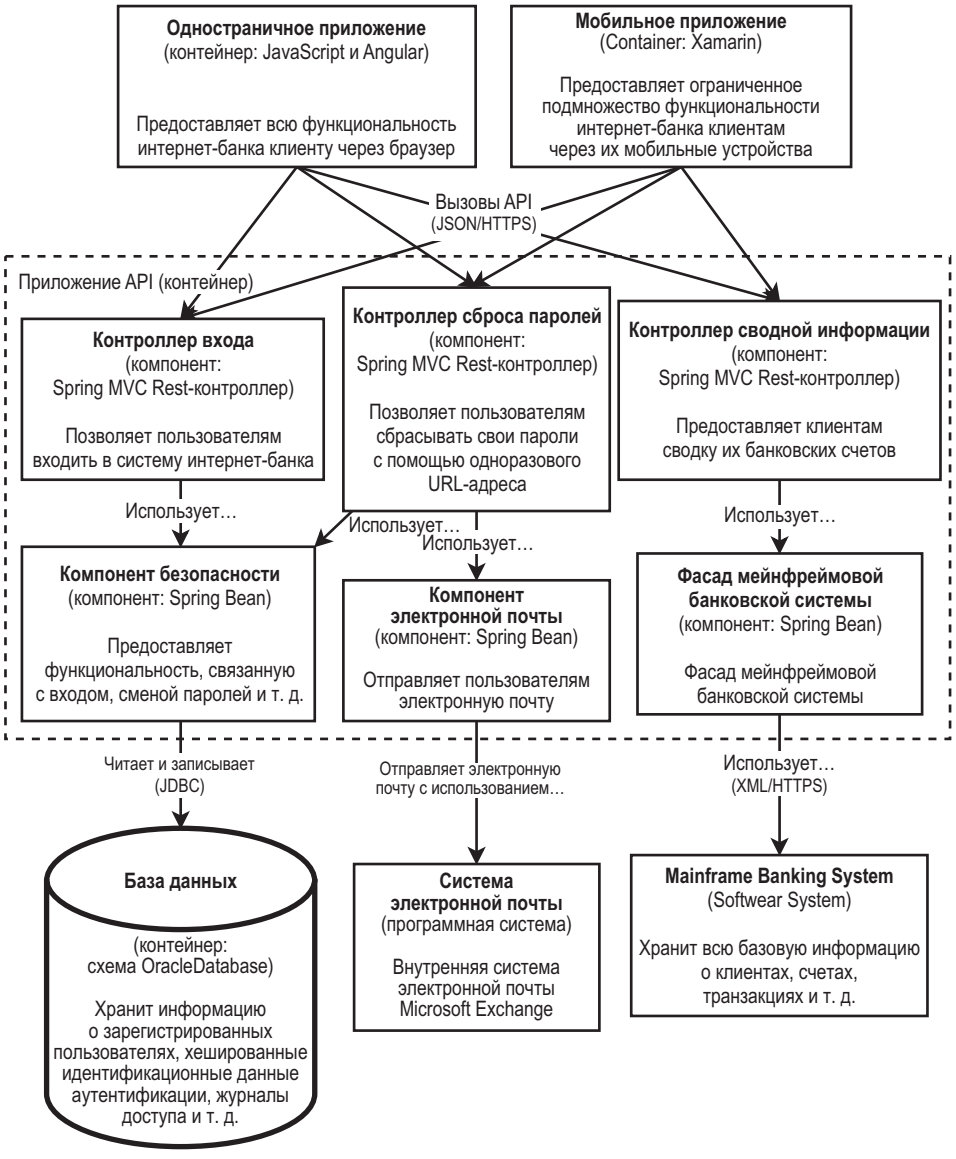

Рис. В.3. Компонентная диаграмма. Изображение с сайта https://c4model.com/, публикуется на условиях лицензии https://creativecommons.org/licenses/by/4.0/

Компонент электронной почты представляет собой клиент, который отправляет запросы к системе электронной почты. Контроллер сброса пароля использует компонент электронной почты для отправки пользователям сообщений для сброса пароля.

Контроллер сводной информации предоставляет пользователям сводку их банковских счетов. Чтобы получить эту информацию, он вызывает функции фасада мейнфреймовой банковской системы, которая, в свою очередь, отправляет запросы к банковской системе. Также в сервисе бэкенда могут присутствовать другие компоненты, не показанные на рис. В.3, которые используют фасад мейнфреймовой банковской системы для отправки запросов к банковской системе.

Кодовая диаграмма представляет собой диаграмму классов UML. (Если вы не знакомы с UML, обратитесь к другим источникам, таким как https://www.uml.org/.) При проектировании интерфейса можно использовать паттерны ООП (объектно-ориентированного программирования).

На рис. В.4 приведен пример кодовой диаграммы фасада мейнфреймовой банковской системы с рис. В.3. В соответствии с паттерном «Фасад», интерфейс `MainframeBankingSystemFacade` реализуется в классе `MainframeBankingSystemFacadeImpl`. Мы применяем паттерн «Фабрика», в котором объект `MainframeBankingSystemFacadeImpl` создает объект `GetBalanceRequest`. Можно применить паттерн «Шаблонный метод» для определения интерфейса `AbstractRequest` и класса `GetBalanceRequest`, определить интерфейс `InternetBankingSystemException` и класс `MainframeBankingSystemException`, определить интерфейс `AbstractResponse` и класс `GetBalanceResponse`. Объект `MainframeBankingSystemFacadeImpl` может использовать пул соединений `BankingSystemConnection` для соединения и отправки запросов к мейнфреймовой банковской системе *и выдачи* объекта исключения `MainframeBankingSystemException` при обнаружении ошибки. (Внедрение зависимостей на рис. В.4 не показано.)

Диаграммы, созданные в ходе собеседования или содержащиеся в документации системы, обычно содержат не компоненты одного конкретного уровня, а набор компонентов уровней 1–3.

Ценность модели C4 заключается не в тщательном перенесении этой структуры на бумагу, а в понимании ее уровней абстракции и умении быстро изменять масштаб структуры системы (как с повышением, так и с понижением) в процессе проектирования системы.

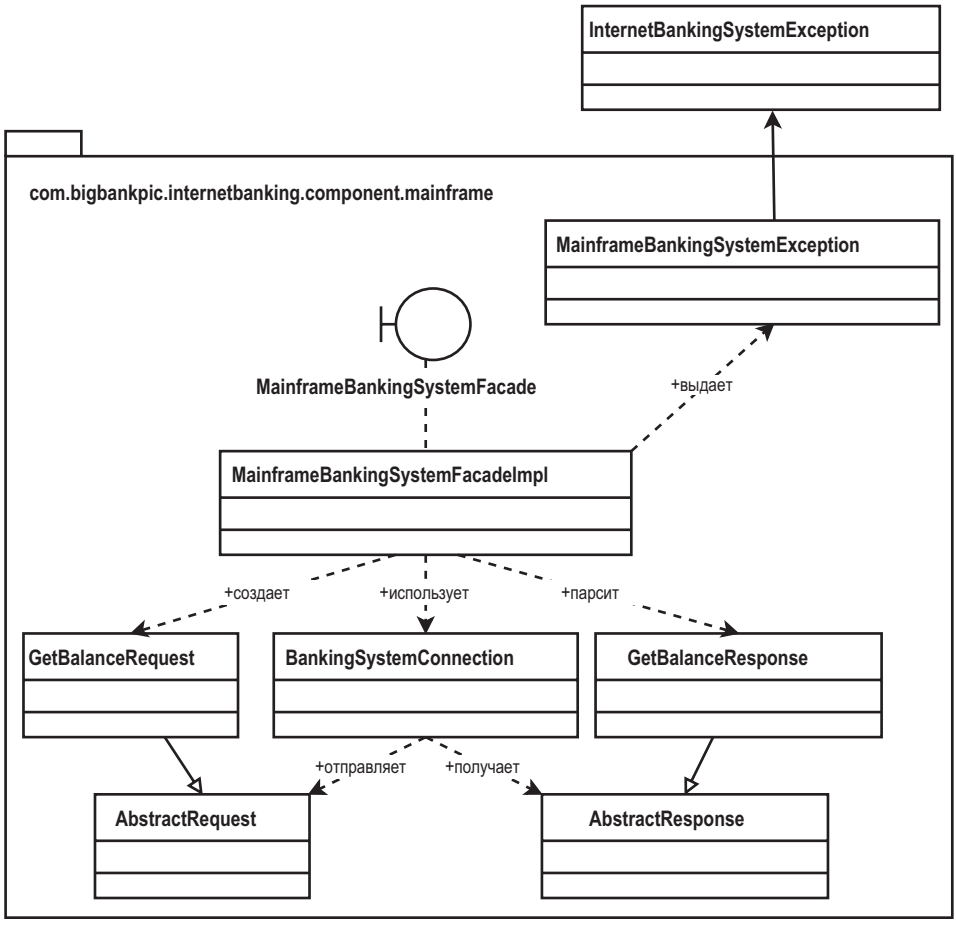

Рис. В.4. Кодовая диаграмма (диаграмма классов UML). Изображение с сайта https://c4model.com/, публикуется на условиях лицензии https://creativecommons.org/licenses/by/4.0/

Г

Двухфазный коммит (2PC)

Двухфазный коммит (2PC) здесь рассматривается как возможный метод управления распределенными транзакциями, но следует подчеркнуть, что он не подходит для распределенных сервисов. Если в ходе собеседования вы затронули распределенные транзакции, можно кратко обсудить двухфазный коммит как потенциальную возможность, а также то, почему его не следует использовать с сервисами.

На рис. Г.1 показано успешное выполнение двухфазного коммита. Двухфазный коммит состоит из двух фаз (отсюда и название): фазы подготовки и фазы коммита. Координатор сначала отправляет запрос на подготовку к каждой базе данных. (Мы называем получателей «базами данных», но это могут быть и сервисы или системы других типов.) Если каждая база данных отвечает успешно, то координатор отправляет запрос на коммит каждой базе данных. Если какая-либо база данных не отвечает или отвечает ошибкой, координатор отправляет всем базам данных запрос на отмену.

Двухфазный коммит обеспечивает согласованность за счет снижения производительности из-за требований блокировки. К его недостаткам можно отнести необходимость присутствия координатора, без которого может возникнуть несогласованность. На рис. Г.2 показано, что сбой координатора в фазе коммита может вызвать несогласованность, так как с некоторыми базами данных коммит будет выполнен, а с некоторыми нет. Более того, недоступность координатора делает невозможными любые операции записи в базы данных.

Если задействованные базы данных не будут ни коммитить, ни отменять транзакции, пока их исход не будет однозначно определен, это позволит предотвратить рассогласование. С другой стороны, это будет означать, что транзакции могут блокировать и останавливать выполнение других транзакций, пока координатор не возобновит работу.

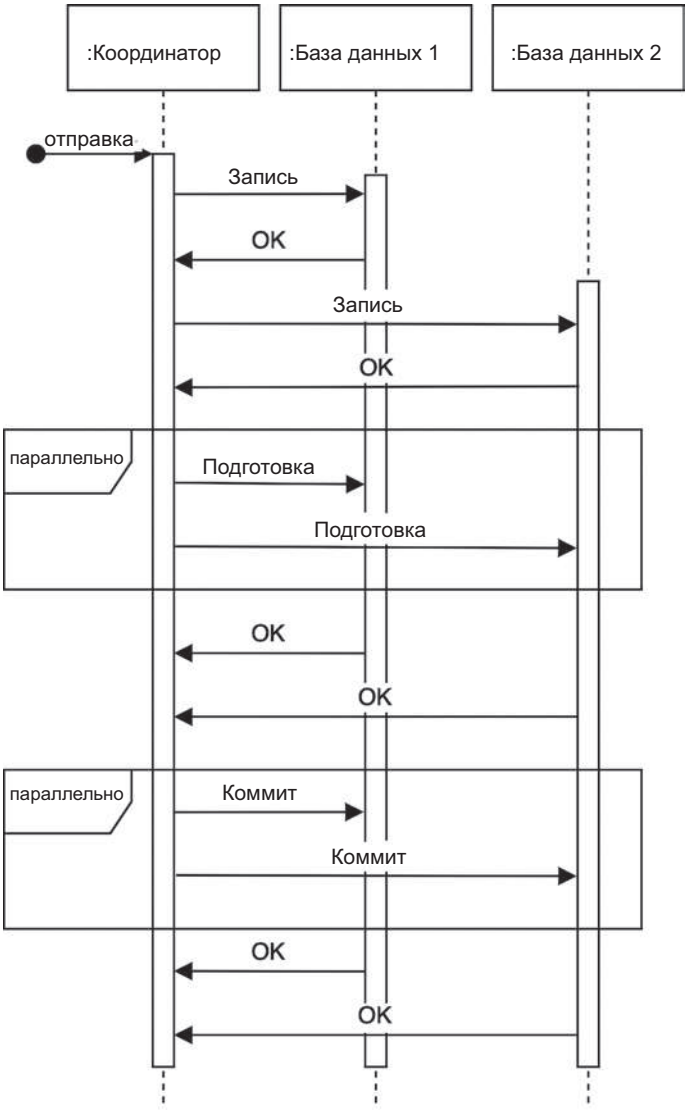

Рис. Г.1. Успешное выполнение 2PC. На схеме изображены две базы данных, но их количество может быть любым. Схема взята из книги Designing Data-Intensive Applications, Martin Kleppmann, 2017, O'Reilly Media

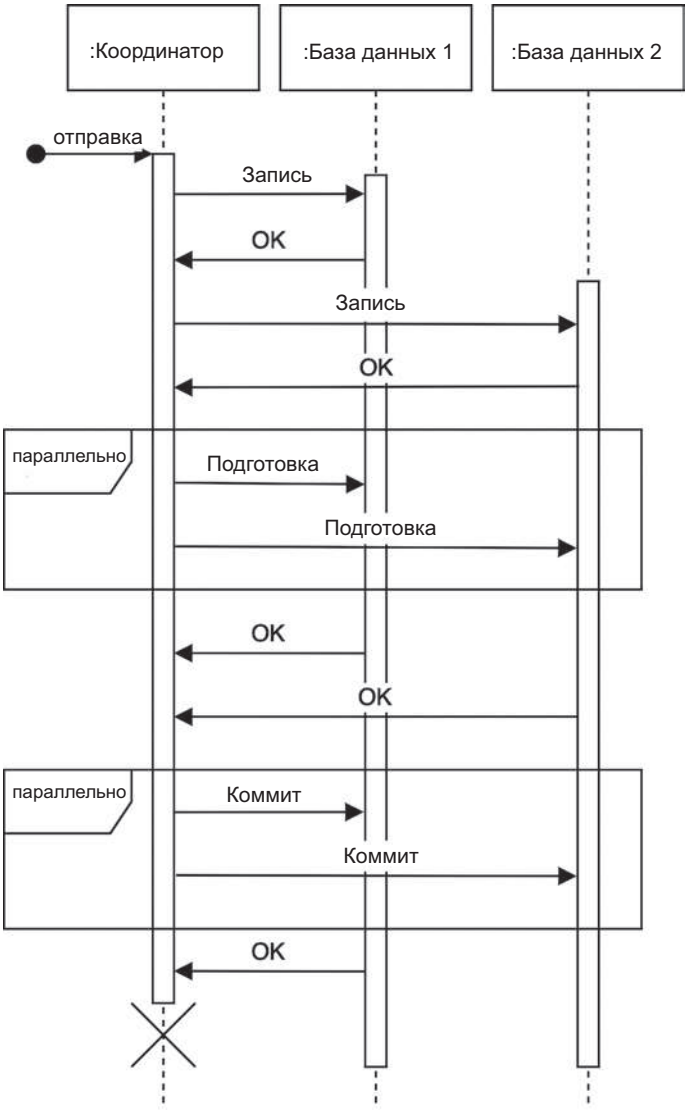

Рис. Г.2. Сбой координатора в фазе коммита приводит к рассогласованию информации. Схема взята из книги Designing Data-Intensive Applications, Martin Kleppmann, 2017, O'Reilly Media

Двухфазный коммит требует общего API для всех баз данных в целях взаимодействия с координатором. Стандарт называется X/Open XA (eXtended Architecture); это API на языке C, который также имеет привязки к другим языкам.

Двухфазный коммит обычно не подходит для сервисов по следующим причинам:

- Координатор должен регистрировать все транзакции, чтобы при восстановлении после сбоя сравнивать свой журнал с базами данных для принятия решения о синхронизации. Это создает дополнительные затраты памяти.
- Кроме того, этот вариант не подходит для сервисов без сохранения состояния, которые могут взаимодействовать по протоколу HTTP, а этот протокол не имеет состояния.
- Чтобы произошел коммит, все базы данных должны вернуть ответ (то есть коммит не происходит, если какая-либо база данных недоступна). Возможность корректного сокращения функциональности отсутствует. В целом при этом ухудшается масштабируемость, производительность и отказоустойчивость.
- Восстановление после сбоев и синхронизацию приходится выполнять вручную, поскольку коммит записи состоялся только в части баз данных.
- Затраты на разработку и обслуживание двухфазного коммита во всех задействованных сервисах/базах данных. Подробности протокола, разработки, конфигурации и развертывания должны координироваться между всеми участвующими командами.
- Многие современные технологии не поддерживают двухфазный коммит. Примеры: базы данных NoSQL (например, Cassandra и MongoDB) и брокеры сообщений (например, Kafka и RabbitMQ).
- Двухфазный коммит снижает доступность, так как все задействованные сервисы должны быть доступны для коммита. У других механизмов распределенных транзакций (таких, как сага) этого требования нет.

В табл. Г.1 приводится краткое сравнение двухфазного коммита с сагой. Для распределенных транзакций, в которых задействованы сервисы, следует избегать двухфазного коммита и отдавать предпочтение таким механизмам, как сага, супервизор транзакций, CDC или контрольные точки.

Таблица Г.1. Двухфазный коммит и сага

Двухфазный коммит	Сага
XA — открытый стандарт, но его реализация может быть привязана к конкретной платформе/поставщику	Универсальность. Обычно реализуется отправкой и потреблением сообщений в топик Kafka (см. главу 5)
Обычно применяется для быстрых транзакций	Обычно применяется для продолжительных транзакций
Требует, чтобы коммит транзакции происходил в одном процессе	Транзакция может быть разбита на несколько шагов

Чжиюн Тань

System Design: пережить интервью

Перевел с английского Е. Матвеев

Научный редактор А. Петраки

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Е. Строганова*
Литературный редактор	*М. Трусковская*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 03.2025. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 14.02.25. Формат 70×100/16. Бумага офсетная. Усл. п. л. 43,860. Тираж 1200. Заказ 0000.

Алекс Сюй

SYSTEM DESIGN. ПОДГОТОВКА К СЛОЖНОМУ ИНТЕРВЬЮ

Интервью по System Design (проектированию ИТ-систем) очень популярны у работодателей, на них легко проверить ваши навыки общения и оценить умение решать реальные задачи.

Пройти такое собеседование непросто, поскольку в проектировании ИТ-систем не существует единственно правильных решений. Речь идет о самых разнообразных реальных системах, обладающих множеством особенностей. Вам могут предложить выбрать общую архитектуру, а потом пройтись по всем компонентам или, наоборот, сосредоточиться на каком-то одном аспекте. Но в любом случае вы должны продемонстрировать понимание и знание системных требований, ограничений и узких мест.

Правильная стратегия и знания являются ключевыми факторами успешного прохождения интервью!

КУПИТЬ

КРОК

СОЗДАЕМ НАСТОЯЩЕЕ,
ИНТЕГРИРУЕМ БУДУЩЕЕ

croc.ru

КРОК — технологический партнер с комплексной экспертизой в области построения и развития инфраструктуры, внедрения информационных систем, разработки программных решений и сервисной поддержки.

Центры компетенций КРОК фокусируются на ключевых отраслевых кластерах — промышленность, финансовый сектор, розничные продажи, муниципальное управление, спорт и культура.

Ежегодно сотни проектов КРОК становятся системообразующими для экономики и социально-культурной сферы.