Часть I
Основы информационных систем

Первые четыре главы этой книги посвящены основным идеям, относящимся ко всем информационным системам, работающим на одной машине или распределенным по кластеру.

- В главе 1 вы познакомитесь с терминологией и общим подходом данной книги. Здесь обсуждается, что мы понимаем под словами *«надежность»* (reliability), *«масштабируемость»* (scalability) и *«удобство сопровождения»* (maintainability), а также как достичь этих целей.
- В главе 2 сравниваются несколько разных моделей данных и языков запросов — наиболее заметных качественных различий баз данных с точки зрения разработчика. Кроме того, мы рассмотрим, насколько различные модели подходят для разных ситуаций.
- В главе 3 мы обратимся к внутреннему устройству подсистем хранения и размещению данных на диске теми или иными СУБД. Различные подсистемы хранения оптимизированы под разные нагрузки, и выбор правильной подсистемы может оказать колоссальное влияние на производительность.
- В главе 4 сравниваются разнообразные форматы кодирования данных (сериализации), особенно их работа в средах с меняющимися требованиями приложений, к которым необходимо со временем адаптировать схемы.

Позже, в части II, мы обратимся к конкретным проблемам распределенных информационных систем.


Разработка высоконагруженных данными приложений
Программирование, масштабирование, поддержка
НАДЕЖНОСТЬ
МАСШТАБИРУЕМОСТЬ
УДОБСТВО СОПРОВОЖДЕНИЯ
НАДЕЖНОСТЬ
Устойчивость к аппаратным и программным сбоям. Человеческий фактор
МАСШТАБИРУЕМОСТЬ
Показатели нагрузки и производительности. Время ожидания, процентили и пропускная способность
УДОБСТВО СОПРОВОЖДЕНИЯ
Удобство эксплуатации, простота и возможность развития
N
W
E
S

1 Надежные, масштабируемые и удобные в сопровождении приложения

> Интернет настолько хорош, что большинство людей считает его природным ресурсом, таким как Тихий океан, а не чем-то сотворенным руками человека. Когда в последний раз в технологии подобного масштаба не было ни одной ошибки?
>
> *Алан Кей[1]. Из интервью Dr. Dobb's Journal (2012)*

Многие приложения сегодня относятся к категории *высоконагруженных данными* (data-intensive), в отличие от *высоконагруженных вычислениями* (compute-intensive). Чистая производительность CPU — просто ограничивающий фактор для этих приложений, а основная проблема заключается в объеме данных, их сложности и скорости изменения.

Высоконагруженное данными приложение (DIA) обычно создается из стандартных блоков, обеспечивающих часто требующуюся функциональность. Например, многим приложениям нужно:

- хранить данные, чтобы эти или другие приложения могли найти их в дальнейшем *(базы данных)*;
- запоминать результат ресурсоемкой операции для ускорения чтения *(кэши)*;
- предоставлять пользователям возможность искать данные по ключевому слову или фильтровать их различными способами *(поисковые индексы)*;

[1] http://www.drdobbs.com/architecture-and-design/interview-with-alan-kay/240003442.

- отправлять сообщения другим процессам для асинхронной обработки *(потоковая обработка)*;
- время от времени «перемалывать» большие объемы накопленных данных *(пакетная обработка)*.

Все описанное выглядит до боли очевидным лишь потому, что *информационные системы* — очень удачная абстракция: мы используем их все время, даже не задумываясь. При создании приложения большинство разработчиков и не помышляют о создании с нуля новой подсистемы хранения, поскольку базы данных — инструмент, отлично подходящий для этой задачи.

Но в жизни не все так просто. Существует множество систем баз данных с разнообразными характеристиками, поскольку у разных приложений — различные требования. Существует много подходов к кэшированию, несколько способов построения поисковых индексов и т. д. При создании приложения необходимо определиться с тем, с помощью каких инструментов и подходов лучше всего решать имеющуюся задачу. Кроме того, иногда бывает непросто подобрать нужную комбинацию инструментов, когда нужно сделать что-то, для чего одного инструмента в отдельности недостаточно.

Эта книга проведет обширный экскурс в страну как теоретических принципов, так и практических аспектов информационных систем, а также возможностей их использования для создания высоконагруженных данными приложений. Мы выясним, что объединяет различные инструменты и отличает их, а также то, как они получат свои характеристики.

В этой главе мы начнем с изучения основ того, чего хотим добиться: надежности, масштабируемости и удобства сопровождения информационных систем. Мы проясним, что означают указанные понятия, обрисуем отдельные подходы к работе с информационными системами и пройдемся по необходимым для следующих глав основам. В следующих главах мы продолжим наш послойный анализ, рассмотрим различные проектные решения, которые целесообразно принять во внимание при работе над DIA.

1.1. Подходы к работе над информационными системами

Обычно мы относим базы данных, очереди, кэши и т. п. к совершенно различным категориям инструментов. Хотя базы данных и очереди сообщений выглядят похожими — и те и другие хранят данные в течение некоторого времени, — паттерны доступа для них совершенно различаются, что означает различные характеристики производительности, а следовательно, очень разные реализации.

Так зачем же валить их все в одну кучу под таким собирательным термином, как *информационные системы*?

В последние годы появилось множество новых инструментов для хранения и обработки данных. Они оптимизированы для множества различных сценариев использования и более не укладываются в обычные категории [1]. Например, существуют хранилища данных, применяемые как очереди сообщений (Redis) и очереди сообщений с соответствующим базам данных уровнем надежности (Apache Kafka). Границы между категориями постепенно размываются.

Кроме того, все больше приложений предъявляют такие жесткие или широкие требования, что отдельная утилита уже не способна обеспечить все их потребности в обработке и хранении данных. Поэтому работа разбивается на отдельные задачи, которые *можно* эффективно выполнить с помощью отдельного инструмента, и эти различные инструменты объединяются кодом приложения.

Например, при наличии управляемого приложением слоя кэширования (путем использования Memcached или аналогичного инструмента) либо сервера полнотекстового поиска (такого как Elasticsearch или Solr), отдельно от БД, синхронизация этих кэшей и индексов с основной базой данных становится обязанностью кода приложения. На рис. 1.1 в общих чертах показано, как все указанное могло бы выглядеть (более подробно мы рассмотрим этот вопрос в следующих главах).

Если для предоставления сервиса объединяется несколько инструментов, то интерфейс сервиса или программный интерфейс приложения (API) обычно скрывает подробности реализации от клиентских приложений. По существу, мы создали новую специализированную информационную систему из более мелких, универсальных компонентов. Получившаяся объединенная информационная система может гарантировать определенные вещи: например, что кэш будет корректно сделан недействительным или обновлен при записи, вследствие чего внешние клиенты увидят непротиворечивые результаты. Вы теперь не только разработчик приложения, но и архитектор информационной системы.

При проектировании информационной системы или сервиса возникает множество непростых вопросов. Как обеспечить правильность и полноту данных, в том числе при внутренних ошибках? Как обеспечить одинаково хорошую производительность для всех клиентов даже в случае ухудшения рабочих характеристик некоторых частей системы? Как обеспечить масштабирование для учета возросшей нагрузки? Каким должен быть хороший API для этого сервиса?

Существует множество факторов, влияющих на конструкцию информационной системы, включая навыки и опыт вовлеченных в проектирование специалистов, унаследованные системные зависимости, сроки поставки, степень приемлемости разных видов риска для вашей компании, законодательные ограничения и т. д. Эти факторы очень сильно зависят от конкретной ситуации.

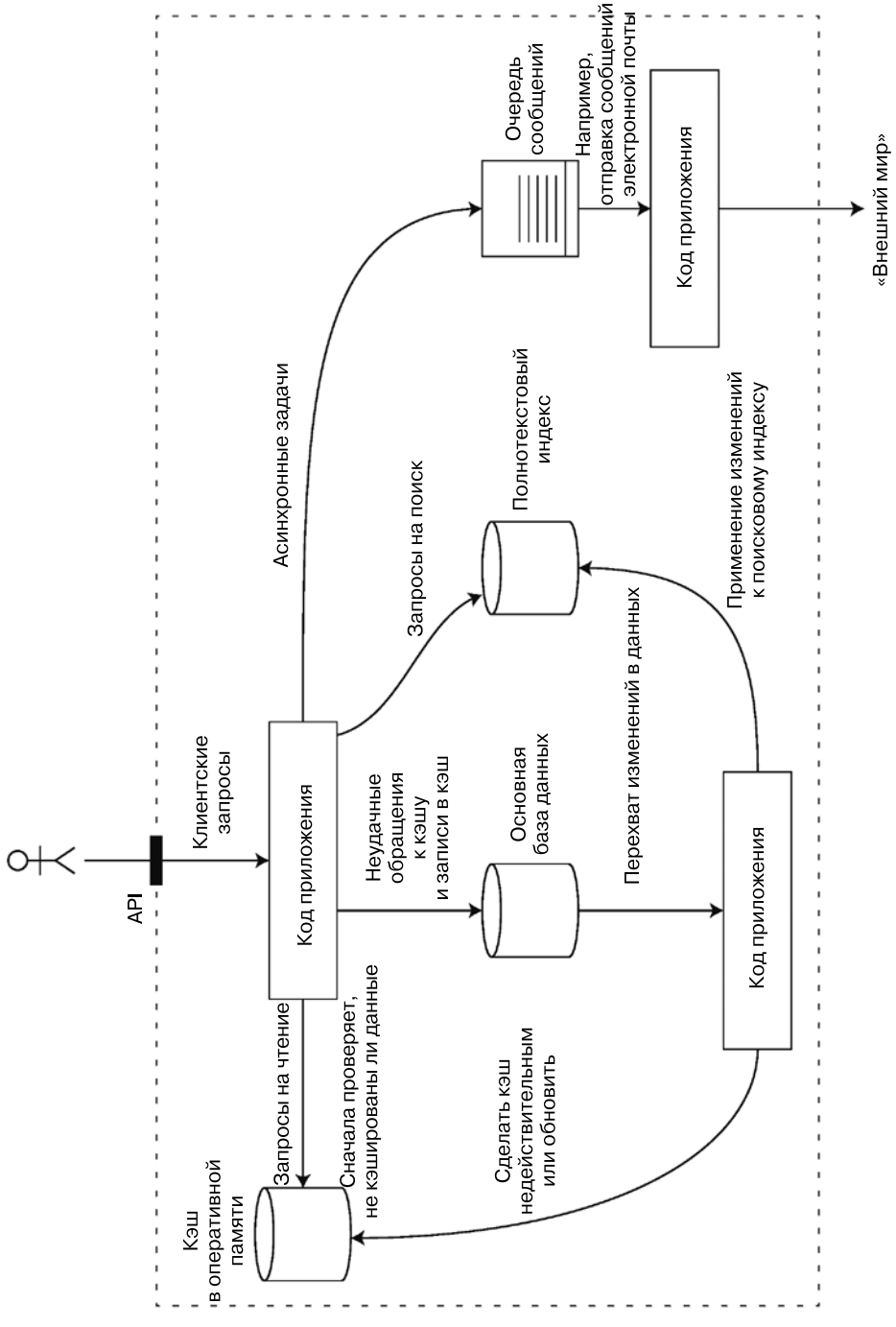

Рис. 1.1. Одна из возможных архитектур информационной системы, состоящей из нескольких компонентов

В данной книге мы сосредоточимся на трех вопросах, имеющих наибольшее значение в большинстве программных систем.

- *Надежность*. Система должна продолжать работать *корректно* (осуществлять нужные функции на требуемом уровне производительности) даже при *неблагоприятных обстоятельствах* (в случае аппаратных или программных сбоев либо ошибок пользователя). См. раздел 1.2.
- *Масштабируемость*. Должны быть предусмотрены разумные способы решения возникающих при *росте* (в смысле объемов данных, трафика или сложности) системы проблем. См. раздел 1.3.
- *Удобство сопровождения*. Необходимо обеспечить возможность эффективной работы с системой множеству различных людей (разработчикам и обслуживающему персоналу, занимающимся как поддержкой текущего функционирования, так и адаптацией системы к новым сценариям применения). См. раздел 1.4.

Приведенные термины часто задействуют без четкого понимания их смысла. Ради более осмысленного проектирования мы потратим остаток главы на изучение концепций надежности, масштабируемости и удобства сопровождения. Далее, в следующих главах, мы рассмотрим различные методики, архитектуры и алгоритмы, используемые для достижения этих целей.

1.2. Надежность

Каждый интуитивно представляет, что значит быть надежным или ненадежным. От программного обеспечения обычно ожидается следующее:

- приложение выполняет ожидаемую пользователем функцию;
- оно способно выдержать ошибочные действия пользователя или применение программного обеспечения неожиданным образом;
- его производительность достаточно высока для текущего сценария использования, при предполагаемой нагрузке и объеме данных;
- система предотвращает любой несанкционированный доступ и неправильную эксплуатацию.

Если считать, что все указанное означает «работать нормально», то термин «надежность» будет иметь значение, грубо говоря, «продолжать работать нормально даже в случае проблем».

Возможные проблемы называются *сбоями*, а системы, созданные в расчете на них, называются *устойчивыми к сбоям*. Этот термин способен ввести в некоторое заблуждение: он наводит на мысль, что можно сделать систему устойчивой ко всем возможным видам сбоев. Однако на практике это неосуществимо. Если наша планета (и все находящиеся на ней серверы) будет поглощена черной дырой, то для обеспечения устойчивости к такому «сбою» потребовалось бы размещение данных

в космосе — удачи вам в одобрении подобной статьи бюджета. Так что имеет смысл говорить об устойчивости лишь к *определенным типам* сбоев.

Обратите внимание, что сбой (fault) и отказ (failure) — разные вещи [2]. Сбой обычно определяется как отклонение одного из компонентов системы от рабочих характеристик, в то время как *отказ* — ситуация, когда вся система в целом прекращает предоставление требуемого сервиса пользователю. Снизить вероятность сбоев до нуля невозможно, следовательно, обычно лучше проектировать механизмы устойчивости к сбоям, которые бы предотвращали переход сбоев в отказы. В этой книге мы рассмотрим несколько методов создания надежных систем из ненадежных составных частей.

Парадоксально, но в подобных устойчивых к сбоям системах имеет смысл *повысить* частоту сбоев с помощью их умышленной генерации — например, путем прерывания работы отдельных, выбранных случайным образом процессов без предупреждения. Многие критические ошибки фактически происходят из-за недостаточной обработки ошибок [3]; умышленное порождение сбоев гарантирует постоянное тестирование механизмов обеспечения устойчивости к ним, что повышает уверенность в должной обработке сбоев при их «естественном» появлении. Пример этого подхода — сервис *Chaos Monkey* компании Netflix [4].

Хотя обычно считается, что устойчивость системы к сбоям важнее их предотвращения, существуют случаи, когда предупреждение лучше лечения (например, когда «лечения» не существует). Это справедливо в случае вопросов безопасности: если атакующий скомпрометировал систему и получил доступ к конфиденциальным данным, то ничего поделать уже нельзя. Однако в этой книге по большей части речь идет о сбоях, допускающих «лечение», как описывается ниже.

Аппаратные сбои

Когда речь идет о причинах отказов систем, первым делом в голову приходят аппаратные сбои. Фатальные сбои винчестеров, появление дефектов ОЗУ, отключение электропитания, отключение кем-то не того сетевого кабеля. Любой, кто имел дело с большими центрами обработки и хранения данных, знает, что подобное происходит *постоянно* при наличии большого количества машин.

Считается, что среднее время наработки на отказ (mean time to failure, MTTF) винчестеров составляет от 10 до 50 лет [5, 6]. Таким образом, в кластере хранения с 10 тысячами винчестеров следует ожидать в среднем одного отказа жесткого диска в день.

Первая естественная реакция на эту информацию — повысить избыточность отдельных компонентов аппаратного обеспечения с целью снизить частоту отказов системы. Можно создать RAID-массивы из дисков, обеспечить дублирование электропитания серверов и наличие в них CPU с возможностью горячей замены,

а также запастись батареями и дизельными генераторами в качестве резервных источников электропитания ЦОДов. При отказе одного компонента его место на время замены занимает резервный компонент. Такой подход не предотвращает полностью отказы, возникающие из-за проблем с оборудованием, но вполне приемлем и часто способен поддерживать бесперебойную работу машин в течение многих лет.

До недавних пор избыточность компонентов аппаратного обеспечения была достаточной для большинства приложений, делая критический отказ отдельной машины явлением вполне редким. При наличии возможности достаточно быстрого восстановления из резервной копии на новой машине время простоя в случае отказа не катастрофично для большинства приложений. Следовательно, многомашинная избыточность требовалась только небольшой части приложений, для которых была критически важна высокая доступность.

Однако по мере роста объемов данных и вычислительных запросов приложений все больше программ начали использовать большее количество машин, что привело к пропорциональному росту частоты отказов оборудования. Более того, на многих облачных платформах, таких как Amazon Web Services (AWS), экземпляры виртуальных машин достаточно часто становятся недоступными без предупреждения [7], поскольку платформы отдают предпочтение гибкости и способности быстро адаптироваться[1] перед надежностью одной машины.

Поэтому происходит сдвиг в сторону систем, способных перенести потерю целых машин, благодаря применению методов устойчивости к сбоям вместо избыточности аппаратного обеспечения или дополнительно к ней. У подобных систем есть и эксплуатационные преимущества: система с одним сервером требует планового простоя при необходимости перезагрузки машины (например, для установки исправлений безопасности), в то время как устойчивая к аппаратным сбоям система допускает установку исправлений по узлу за раз, без вынужденного бездействия всей системы (*плавающее обновление*; см. главу 4).

Программные ошибки

Обычно считают так: аппаратные сбои носят случайный характер и независимы друг от друга: отказ диска одного компьютера не означает, что диск другого скоро тоже начнет сбоить. Конечно, возможны слабые корреляции (например, вследствие общей причины наподобие температуры в серверной стойке), но в остальных случаях одновременный отказ большого количества аппаратных компонентов маловероятен.

Другой класс сбоев — систематическая ошибка в системе [8]. Подобные сбои сложнее предотвратить, и в силу их корреляции между узлами они обычно вызывают

[1] Определение этому термину дается в подразделе «Как справиться с нагрузкой» раздела 1.3.

гораздо больше системных отказов, чем некоррелируемые аппаратные сбои [5]. Рассмотрим примеры.

- Программная ошибка, приводящая к фатальному сбою экземпляра сервера приложения при конкретных «плохих» входных данных. Например, возьмем секунду координации 30 июня 2012 года, вызвавшую одновременное зависание множества приложений из-за ошибки в ядре операционной системы Linux [9].
- Выходит из-под контроля процесс, полностью исчерпавший какой-либо общий ресурс: время CPU, оперативную память, пространство на диске или полосу пропускания сети.
- Сервис, от которого зависит работа системы, замедляется, перестает отвечать на запросы или начинает возвращать испорченные ответы.
- Каскадные сбои, при которых крошечный сбой в одном компоненте вызывает сбой в другом компоненте, а тот, в свою очередь, вызывает дальнейшие сбои [10].

Ошибки, вызывающие подобные программные сбои, часто долго остаются неактивными, вплоть до момента срабатывания под влиянием необычных обстоятельств. При этом оказывается, что приложение делает какое-либо допущение относительно своего окружения — и хотя обычно такое допущение справедливо, в конце концов оно становится неверным по какой-либо причине [11].

Быстрого решения проблемы систематических ошибок в программном обеспечении не существует. Может оказаться полезным множество мелочей, таких как: тщательное обдумывание допущений и взаимодействий внутри системы; всестороннее тестирование; изоляция процессов; предоставление процессам возможности перезапуска после фатального сбоя; оценка, мониторинг и анализ поведения системы при промышленной эксплуатации. Если система должна обеспечивать выполнение какого-либо условия (например, в очереди сообщений количество входящих сообщений должно быть равно количеству исходящих), то можно организовать постоянную самопроверку во время работы и выдачу предупреждения в случае обнаружения расхождения [12].

Человеческий фактор

Проектируют и создают программные системы люди; ими же являются и операторы, обеспечивающие их функционирование. Даже при самых благих намерениях люди ненадежны. Например, одно исследование крупных интернет-сервисов показало, что основной причиной перебоев в работе были допущенные операторами ошибки в конфигурации, в то время как сбои аппаратного обеспечения (серверов или сети) играли какую-либо роль лишь в 10–25 % случаев [13].

Как же обеспечить надежность нашей системы, несмотря на ненадежность людей? Оптимальные системы сочетают в себе несколько подходов.

- Проектирование систем таким образом, который минимизировал бы возможности появления ошибок. Например, грамотно спроектированные абстрак-

ции, API и интерфейсы администраторов упрощают «правильные» действия и усложняют «неправильные». Однако если интерфейсы будут слишком жестко ограничены, то люди начнут искать пути обхода; это приведет к нивелированию получаемой от таких интерфейсов выгоды, так что самое сложное здесь — сохранить равновесие.

- Расцепить наиболее подверженные человеческим ошибкам места системы с теми местами, где ошибки могут привести к отказам. В частности, предоставить не для промышленной эксплуатации полнофункциональную среду-*«песочницу»*, в которой можно было бы безопасно изучать работу и экспериментировать с системой с помощью настоящих данных, не влияя на реальных пользователей.
- Выполнять тщательное тестирование на всех уровнях, начиная с модульных тестов и заканчивая комплексным тестированием всей системы и ручными тестами [3]. Широко используется автоматизированное тестирование, вполне приемлемое и особенно ценное для пограничных случаев, редко возникающих при нормальной эксплуатации.
- Обеспечить возможность быстрого и удобного восстановления после появления ошибок для минимизации последствий в случае отказа. Например, предоставить возможность быстрого отката изменений конфигурации, постепенное внедрение нового кода (чтобы все неожиданные ошибки оказывали влияние на небольшое подмножество пользователей) и возможность использования утилит для пересчета данных (на случай, если окажется, что предыдущие вычисления были неправильными).
- Настроить подробный и ясный мониторинг, в том числе метрик производительности и частот ошибок. В других областях техники это носит название *телеметрии*. (После отрыва ракеты от земли телеметрия превращается в важнейшее средство отслеживания происходящего и выяснения причин отказов [14].) Мониторинг обеспечивает диагностические сигналы на ранних стадиях и позволяет проверять, не были ли нарушены правила или ограничения. При возникновении проблемы метрики оказываются бесценным средством диагностики проблем.
- Внедрение рекомендуемых управленческих практик и обучение — сложный и важный аспект, выходящий за рамки данной книги.

Насколько важна надежность?

Надежность нужна не только в управляющем программном обеспечении атомных электростанций и воздушного сообщения — от обычных приложений тоже ожидается надежная работа. Ошибки в коммерческих приложениях приводят к потерям производительности (и юридическим рискам, если цифры в отчетах оказались неточны), а простои сайтов интернет-магазинов могут приводить к колоссальным убыткам в виде недополученных доходов и ущерба для репутации.

Создатели даже «некритичных» приложений несут ответственность перед пользователями. Возьмем, например, родителей, хранящих все фотографии и видео своих детей в вашем приложении для фото [15]. Как они будут себя чувствовать,

если база данных неожиданно окажется испорченной? Смогут ли они восстановить данные из резервной копии?

Встречаются ситуации, в которых приходится пожертвовать надежностью ради снижения стоимости разработки (например, при создании прототипа продукта для нового рынка) или стоимости эксплуатации (например, для сервиса с очень низкой маржой прибыли) — но «срезать углы» нужно очень осторожно.

1.3. Масштабируемость

Даже если на сегодняшний момент система работает надежно, нет гарантий, что она будет так же работать в будущем. Одна из частых причин снижения эффективности — рост нагрузки: например, система выросла с 10 тыс. работающих одновременно пользователей до 100 тыс. или с 1 до 10 млн. Это может быть и обработка значительно бóльших объемов данных, чем ранее.

Масштабируемость (scalability) — термин, который я буду использовать для описания способности системы справляться с возросшей нагрузкой. Отмечу, однако, что это не одномерный ярлык, который можно «навесить» на систему: фразы «X — масштабируемая» или «Y — немасштабируемая» бессмысленны. Скорее, обсуждение масштабирования означает рассмотрение следующих вопросов: «Какими будут наши варианты решения проблемы, если система вырастет определенным образом?» и «Каким образом мы можем расширить вычислительные ресурсы в целях учета дополнительной нагрузки?».

Описание нагрузки

Во-первых, нужно сжато описать текущую нагрузку на систему: только тогда мы сможем обсуждать вопросы ее роста («Что произойдет, если удвоить нагрузку?»). Нагрузку можно описать с помощью нескольких чисел, которые мы будем называть *параметрами нагрузки*. Оптимальный выбор таких параметров зависит от архитектуры системы. Это может быть количество запросов к веб-серверу в секунду, отношение количества операций чтения к количеству операций записи в базе данных, количество активных одновременно пользователей в комнате чата, частота успешных обращений в кэш или что-то еще. Возможно, для вас будет важно среднее значение, а может, узкое место в вашей ситуации будет определяться небольшим количеством предельных случаев.

Чтобы прояснить эту идею, рассмотрим в качестве примера социальную сеть Twitter, задействуя данные, опубликованные в ноябре 2012 года [16]. Две основные операции сети Twitter таковы:

- *публикация твита* — пользователь может опубликовать новое сообщение для своих подписчиков (в среднем 4600 з/с, пиковое значение — более 12 000 з/с);
- *домашняя лента* — пользователь может просматривать твиты, опубликованные теми, на кого он подписан (300 000 з/с).

Просто обработать 12 тысяч записей в секунду (пиковая частота публикации твитов) должно быть несложно. Однако проблема с масштабированием сети Twitter состоит не в количестве твитов, а в *коэффициенте разветвления по выходу*[1] — каждый пользователь подписан на множество людей и, в свою очередь, имеет много подписчиков. Существует в общих чертах два способа реализации этих двух операций.

1. Публикация твита просто приводит к вставке нового твита в общий набор записей. Когда пользователь отправляет запрос к своей домашней ленте, выполняется поиск всех людей, на которых он подписан, поиск всех твитов для каждого из них и их слияние (с сортировкой по времени). В реляционной базе данных, такой как на рис. 1.2, это можно выполнить путем следующего запроса:

```
SELECT tweets.*, users.* FROM tweets
  JOIN users   ON tweets.sender_id    = users.id
  JOIN follows ON follows.followee_id = users.id
  WHERE follows.follower_id = current_user
```

2. Поддержка кэша для домашней ленты каждого пользователя — аналог почтового ящика твитов для каждого получателя (рис. 1.3). Когда пользователь *публикует твит*, выполняется поиск всех его подписчиков и вставка нового твита во все кэши их домашних лент. Запрос на чтение домашней ленты при этом становится малозатратным, поскольку его результат уже был заранее вычислен.

Первая версия социальной сети Twitter использовала подход 1, но система еле справлялась с нагрузкой от запросов домашних лент, вследствие чего компания переключилась на подход 2. Этот вариант работал лучше, поскольку средняя частота публикуемых твитов почти на два порядка ниже, чем частота чтения домашних лент, так что в данном случае предпочтительнее выполнять больше операций во время записи, а не во время чтения.

Однако недостатком подхода 2 является необходимость в значительном количестве дополнительных действий для публикации твита. В среднем твит выдается 75 подписчикам, поэтому 4,6 тысяч твитов в секунду означает 345 тысяч записей в секунду в кэши домашних лент. Но приведенное здесь среднее значение маскирует тот факт, что количество подписчиков на пользователя сильно варьируется, и у некоторых пользователей насчитывается более 30 миллионов подписчиков[2]. То есть один твит может привести к более чем 30 миллионов записей в домашние ленты! Выполнить их своевременно — Twitter пытается выдавать твиты подписчикам в течение пяти секунд — непростая задача.

---

[1] Термин, заимствованный из электроники, где описывает число логических вентилей, чьи входы подключаются к выходу данного вентиля. Выход должен обеспечивать ток, достаточный для работы всех подключенных входов. В системах обработки транзакций термин используется для описания числа запросов к другим сервисам, которое нужно выполнить, чтобы обслужить один входящий запрос.

[2] Этот рекорд давно побит, количество подписчиков у певицы Кэти Перри в июне 2017 года превысило 100 миллионов — *Примеч. пер.*