Урок 7 > Масштабирование системы

> Оглавление

> Балансировка нагрузки

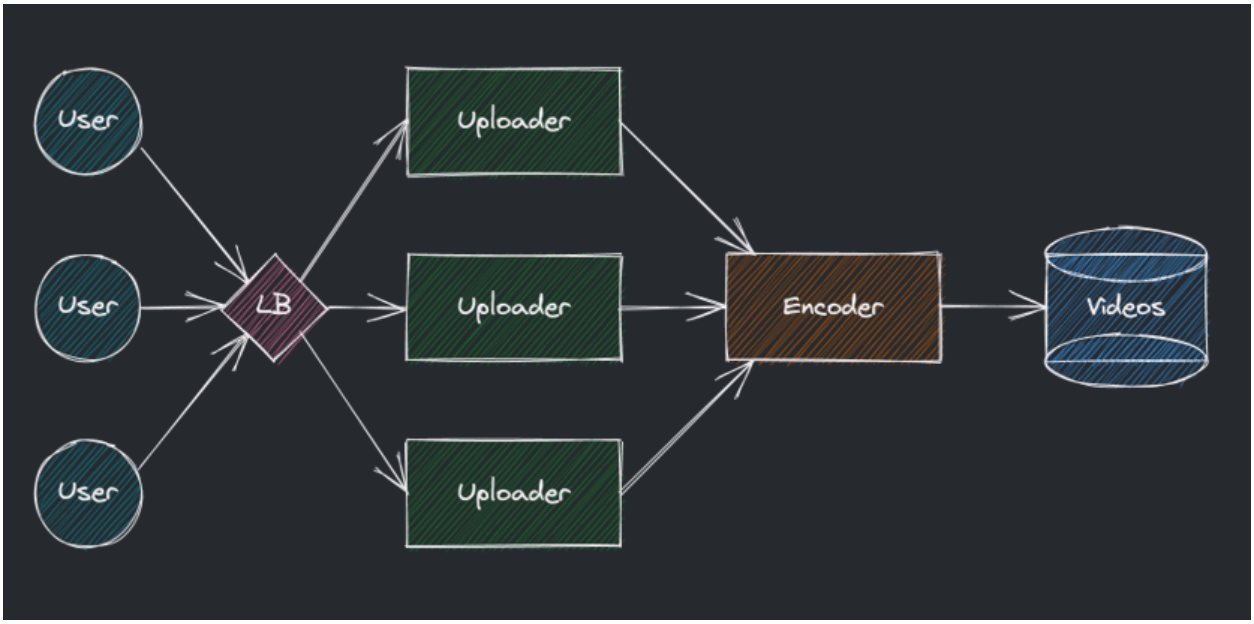
Балансировка нагрузки — это метод распределения заданий между несколькими сетевыми устройствами (например, серверами) с целью оптимизации использования ресурсов, сокращения времени обслуживания запросов, горизонтального масштабирования кластера (динамическое добавление/удаление устройств), а также обеспечения отказоустойчивости (резервирования).

В компьютерах балансировка нагрузки распределяет нагрузку между несколькими вычислительными ресурсами, такими как компьютеры, компьютерные кластеры, сети, центральные процессоры или диски. Цель балансировки нагрузки — оптимизация использования ресурсов, максимизация пропускной способности, уменьшение времени отклика и предотвращение перегрузки какого-либо одного ресурса. Использование нескольких компонентов балансировки нагрузки вместо одного компонента может повысить надежность и доступность за счет резервирования. Балансировка нагрузки предполагает обычно наличие специального программного обеспечения или аппаратных средств, таких как многоуровневый коммутатор или система доменных имен, как серверный процесс.

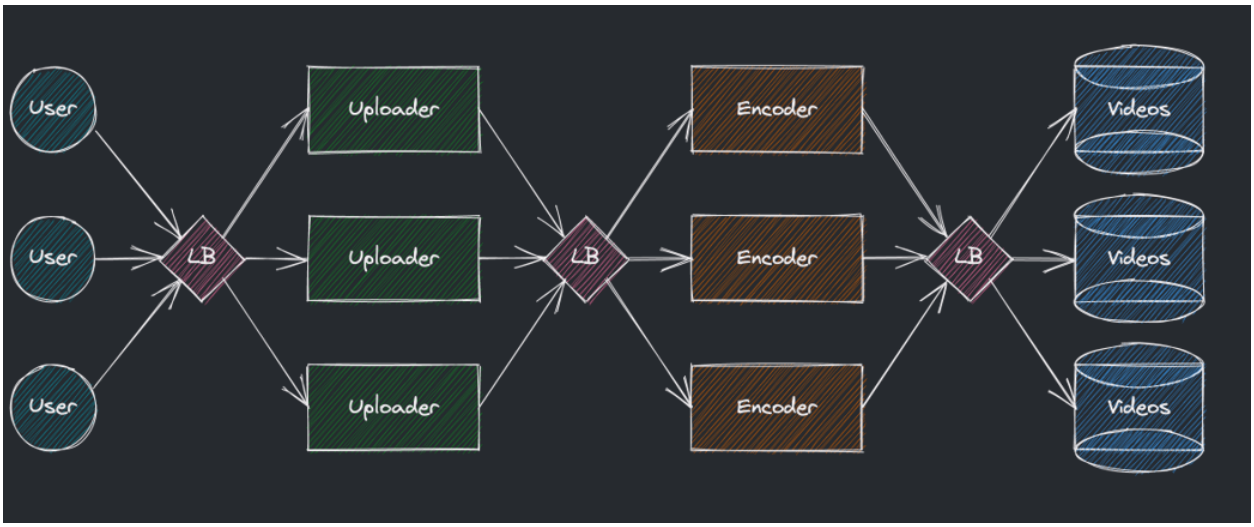
Балансировщик нагрузки (load balancer) – критически важный компонент любой распределенной системы, который распределяет передаваемый трафик между разными приложениями, сервисами и хранилищами.

Балансировщик также может следить за статусом сервисов, в которые он передает запросы, чтобы перестать передавать запросы к переставшим отвечать в течение заданного времени серверам.

Естественной задачей для балансировщика является балансировка пользовательского трафика — распределение приходящих запросов от пользователя между инстансами сервиса:



Но для максимальной надежности они могут располагаться между каждой парой уровней логики системы:



Принцип работы балансировщика нагрузки

- Анализ того, какие из серверов готовы принять и обработать запрос пользователя
- Выбор наиболее подходящего сервера по заданному алгоритму

Алгоритм выбора подходящего сервера

- По количеству подключений — приоритет серверу с минимальным количеством подключений
- По времени ответа — приоритет наименьшей задержке перед ответом
- По трафику — приоритет серверу с минимальной сетевой нагрузкой
- round-robin на каждый запрос меняем сервер по кругу из заранее определенной очереди
- Weighted round-robin — серверы получают веса на основе комбинации из характеристик выше
- Пользовательский хэш — выбираем сервер на основе хэша (IP, user_id, ...)

Преимущества использования балансировщика

- Улучшается пользовательский опыт за счет передачи запроса подходящему серверу
- Снижается нагрузка на каждый из серверов за счет распределения запросов между ними
- Справляемся с отказом части серверов, по крайней мере, в случае stateless сервиса
- Умные балансировщики могут даже предсказывать рост нагрузки и инициировать автоскейлинг

> Распределение данных

При разрастании популярности сервиса растет и объем данных, которые необходимо хранить, к которым обеспечиваем быстрый и надежный доступ множеству пользователей. Для распределения данных есть несколько основных подходов:

- Секционирование (partitioning, партиционирование) – разделение больших хранимых файлов или объектов в базах данных, таких как таблицы, индексы и пр. на логические части по каким-то критериям. Секционирование повышает масштабируемость и производительность системы.
- Избыточность и репликация. Под избыточностью понимается дублирование важных элементов системы для повышения надежности. Под репликацией понимается копирование данных в избыточные сервисы для консистентности.

> Секционирование

Выделяют несколько видов секционирования:

- Вертикальное (по умолчанию) секционирование подразумевает выделение разных смысловых частей данных и последующее вероятное разнесение их по

разным машинам, запас тем самым ограничен.

- Горизонтальное секционирование, оно же шардирование (sharding) подразумевает разбиение данных на уровне таблиц по выбранному ключу, например дню недели или почтовому индексу.

Способы секционирования записей

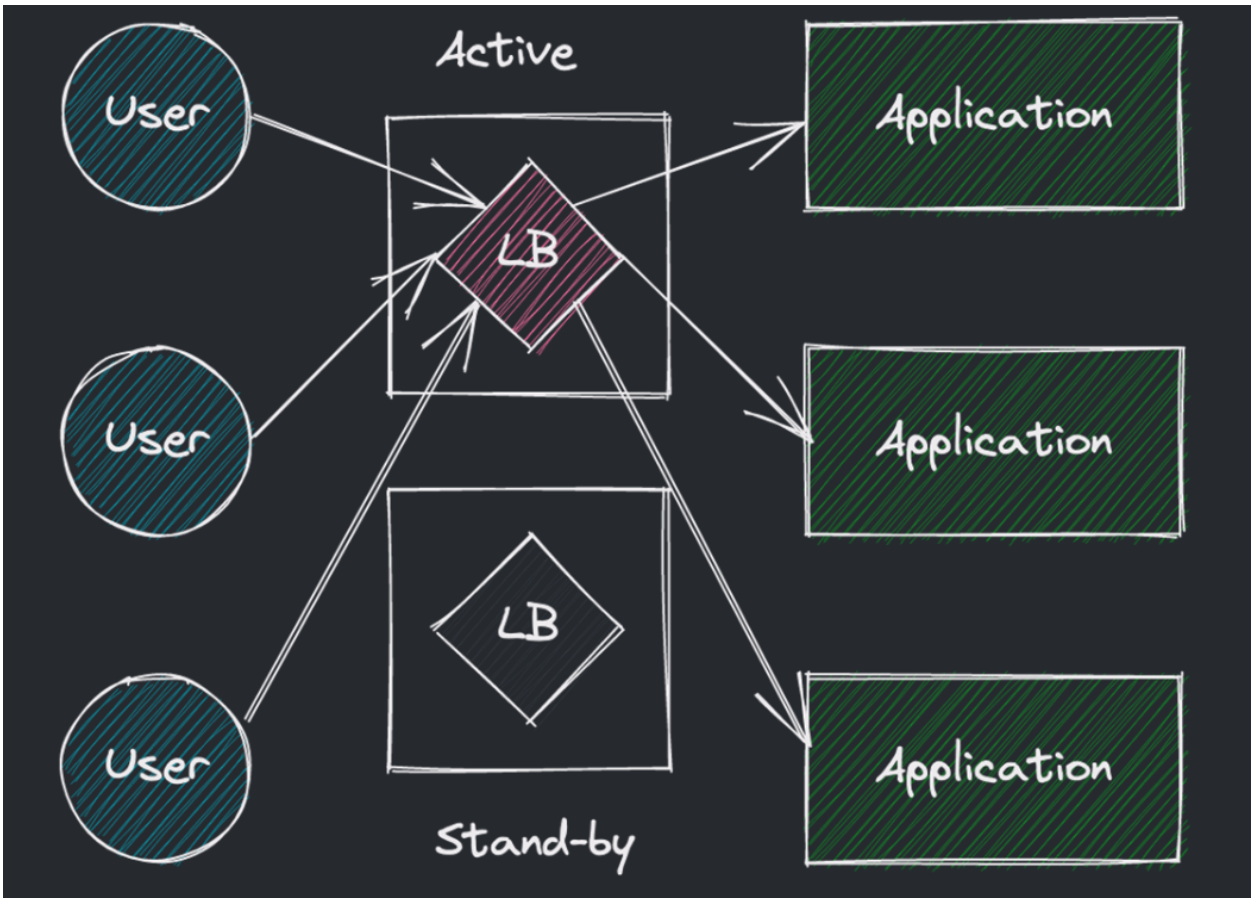
- На основе ключа или хеша: применяем хеш-функцию к какому-либо из полей и получаем номер партиции. Например, если мы располагаем сотней серверов и сохраняем время создания записей как int, то номер партиции можно определить по формуле «timestamp % 100». При удачном выборе такого ключа (не weekday % 7) данные будут распределены предопределённым образом и равномерно. Из минусов – фиксированное количество серверов. Выпадет один и начнется перекос в нагрузке.
- По спискам – мы можем заранее распределить все поступающие записи по предопределенным группам на основе страны, дня недели и т.п., понимая, как соотносится нагрузка из разных групп.
- Round-robin разбиение – меняем партиции по кругу с каждой новой поступающей записью.
- Составное разбиение – комбинируем что-либо из вышеупомянутого.

Недостатки метода

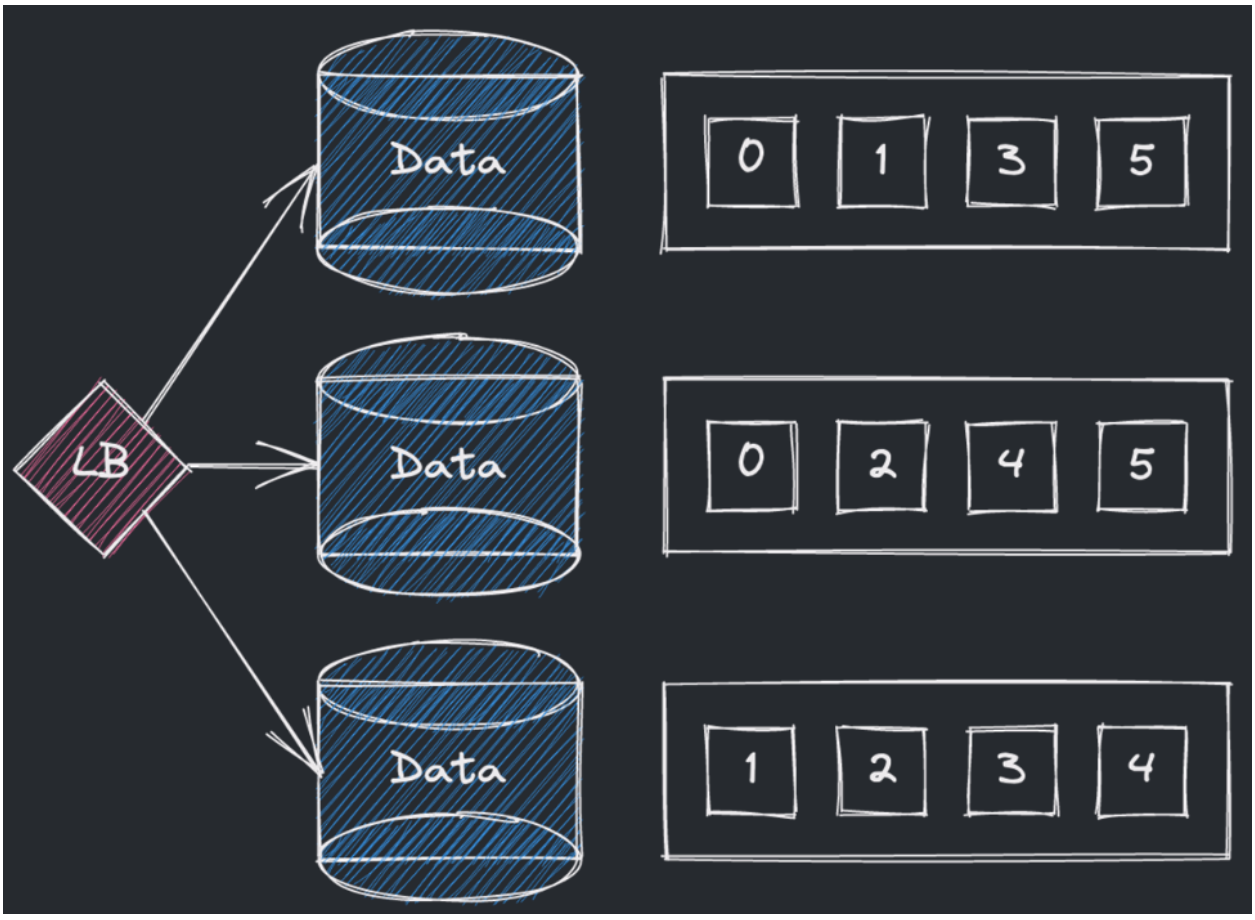
- Выполнение JOIN-ов – они теперь затрагивают данные на нескольких серверах и часто невыполнимы. Частично решается денормализацией БД.
- Большинство РСУБД не могут следить за валидностью внешних ключей на разных серверах.
- Ребалансировка – иногда нам нужно изменить способ шардирования, если поздно поняли уязвимость текущего способа, и как следствие получили неравномерность данных или запросов по серверам. Скорее всего это приведет к пересылке всех данных между серверами и недоступности все это время.

> Избыточность и репликация

Избыточность



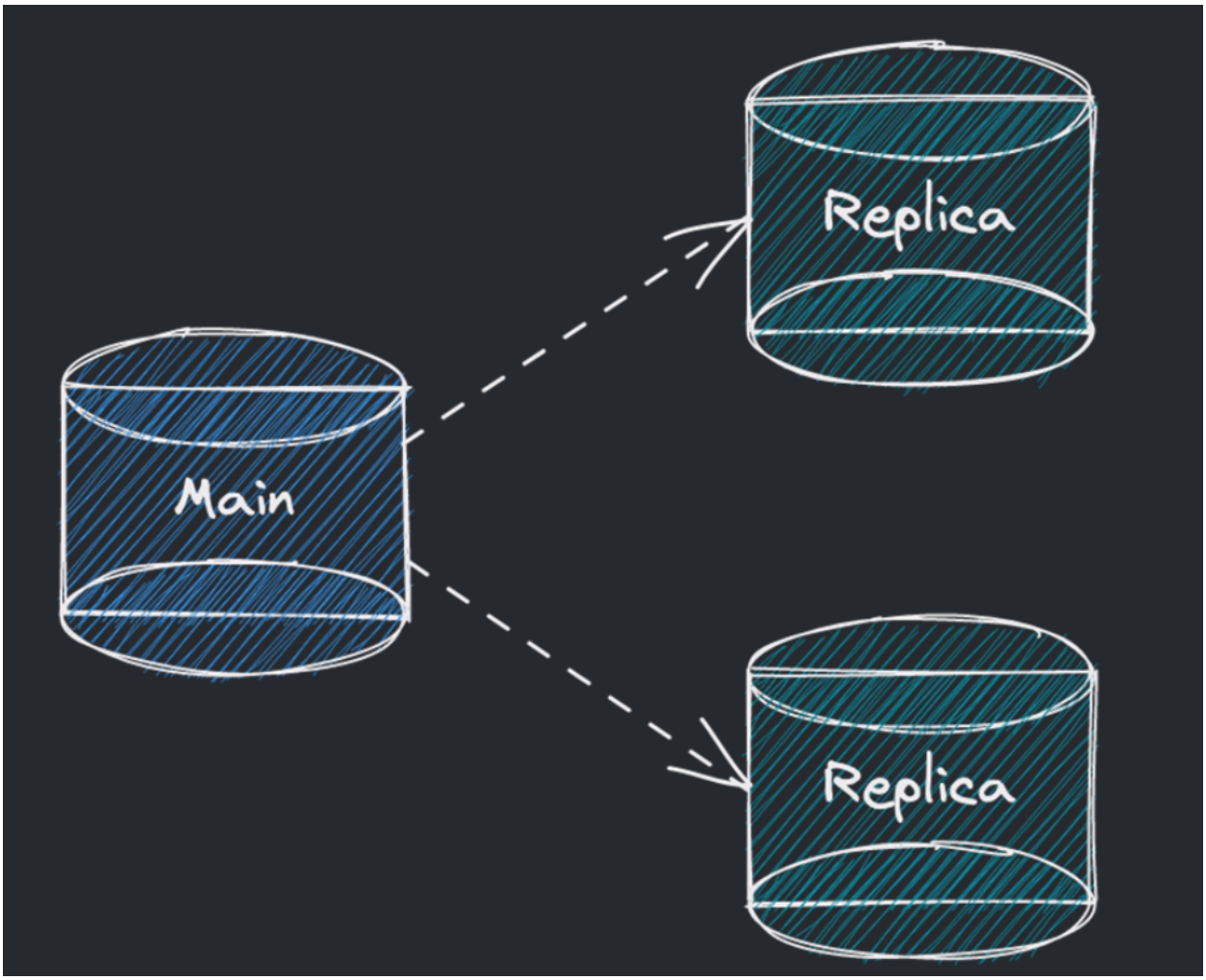
Резервный сервис в пару к действующему на отдельной машине



Размноженные копии файлов с заданным коэффициентом репликации.

Репликация

Репликация повышает надежность, устойчивость к отказам и доступность системы.



При репликации баз данных у нас есть множество разных копий одних и тех же данных.

Но как гарантировать их консистентность для всех пользователей? Есть две стратегии:

- Кворум
- Лидер и последователи

Кворум

Некий состоящий из ⌊N/2⌋ + 1 узлов уровень согласованности носит название кворума, большинства узлов. В случае разбиения сетевой среды на части или отказа узлов, в системе с `2f + 1` узлами, живые узлы способны продолжать приём чтений и записей при недоступности до `f` узлов до тех пор,

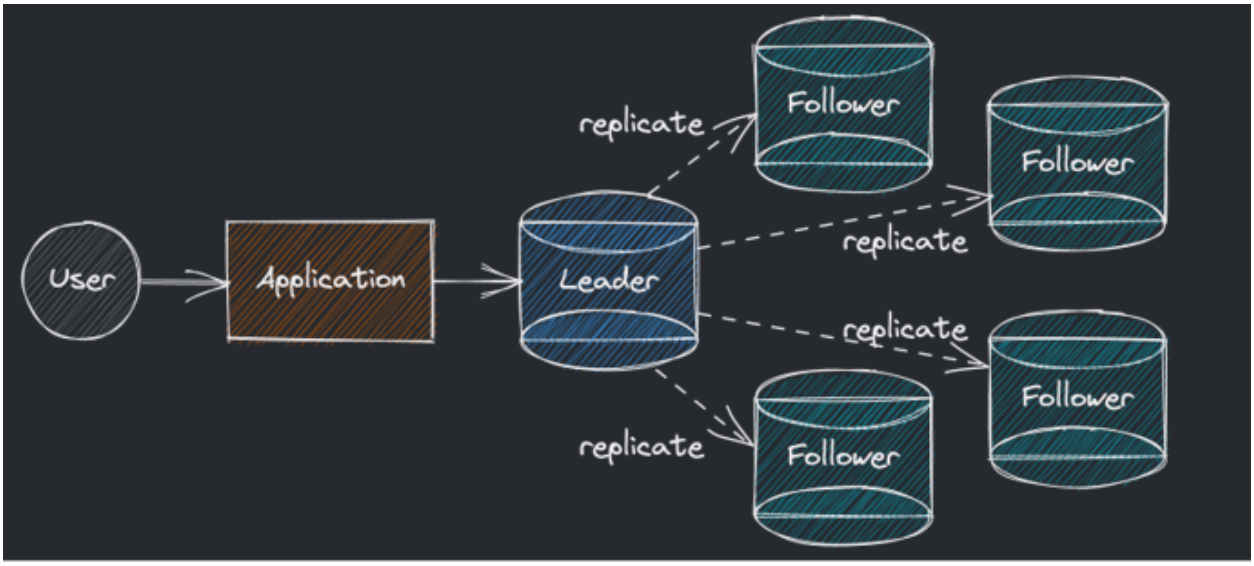
пока оставшаяся часть кластера не станет снова доступной. Иными словами, такая система безразлична к отказу не более `f` узлов.

Выполняя операцию чтения и записи с применением кворума, система не может быть безразлична к отказу большинства узлов. Например, когда в сумме имеются три реплики, а две из них упали, операции чтения и записи не способны достичь значения числа узлов, требуемого для согласованности чтения и записи, ибо лишь один узел из трёх способен отвечать на соответствующие запросы.

Достижение кворума как критерий успешности операции гарантирует консистентность данных. Тем не менее необходимость достижения кворума влечет и эпизодическую недоступность сервиса.

Лидер и последователи

В каждый момент времени есть только один лидер, который осуществляет репликацию данных. Последователи пишут данные по запросу лидера, служат резервной копией и помогают при чтении. В случае отказа лидера один из последователей становится новым лидером:

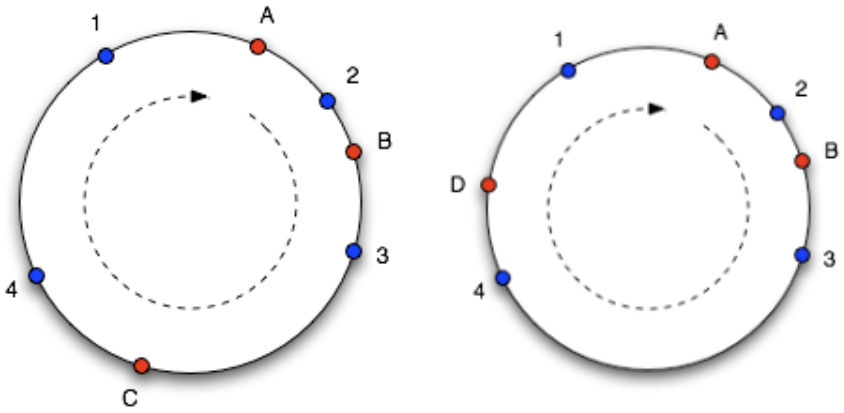


Лидер и последователи

> Консистентное (согласованное) хеширование

При использовании стандартных методов распределения данных между серверами может возникнуть ситуация, когда один или несколько серверов становятся недоступны. Тогда приходится перестраивать логику распределения данных. Такой подход очень ресурсоемкий и не позволяет добиться хороших показателей по скорости ответа системы. С данной проблемой может помочь подход, называемый согласованным или консистентным хешированием.

Согласованное хеширование (англ. consistent hashing) — особый вид хеширования, отличающийся тем, что когда хеш-таблица перестраивается, только K/n ключей в среднем должны быть переназначены, где K — число ключей и n число слотов (slots, buckets). В противоположность этому, в большинстве традиционных хеш-таблиц, изменение количества слотов вызывает переназначение почти всех ключей.



Суть алгоритма заключается в следующем: мы рассматриваем набор целых чисел от 0 до 2^32, «закручивая» числовую ось в кольцо (склеиваем 0 и 2^32). Каждому серверу из пула серверов мы сопоставляем число на кольце (рисунок слева, сервера A, B и C). Ключ хешируется в число в том же диапазоне (на рисунке – синие точки 1-4), в качестве сервера для хранения ключа мы выбираем сервер в точке, ближайшей к точке ключа в направлении по часовой стрелке. Если сервер удаляется из пула или добавляется в пул, на оси появляется или исчезает точка сервера, в результате чего лишь часть ключей перемещается на другой сервер. На рисунке 2 справа показана ситуация, когда сервер C был удалён из пула серверов

и добавлен новый сервер D. Легко заметить, что ключи 1 и 2 не поменяли привязки к серверам, а ключи 3 и 4 переместились на другие сервера. На самом деле одному серверу ставится в соответствие 100-200 точек на оси (пропорционально его весу в пуле), что улучшает равномерность распределения ключей по серверам в случае изменения их конфигурации. В случае консистентного хеширования ключей ранее рассмотренный пример будет выглядеть так, что ключи которые были на 3х серверах и остались в рабочем состоянии, на них же и останутся. А ключи с 4-го сервера равномерно распределяться по 3-м оставшимся, мы потеряли ровно 25% ключей — возможный минимум.