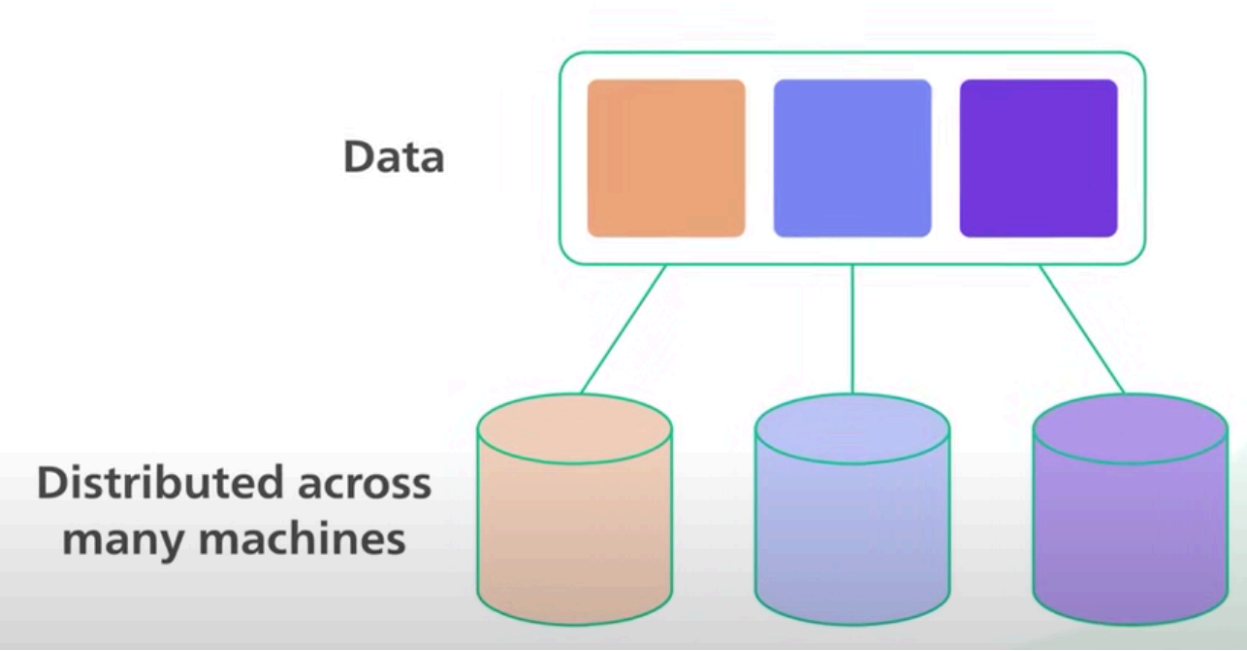
Consistent hashing

▼ Заметки

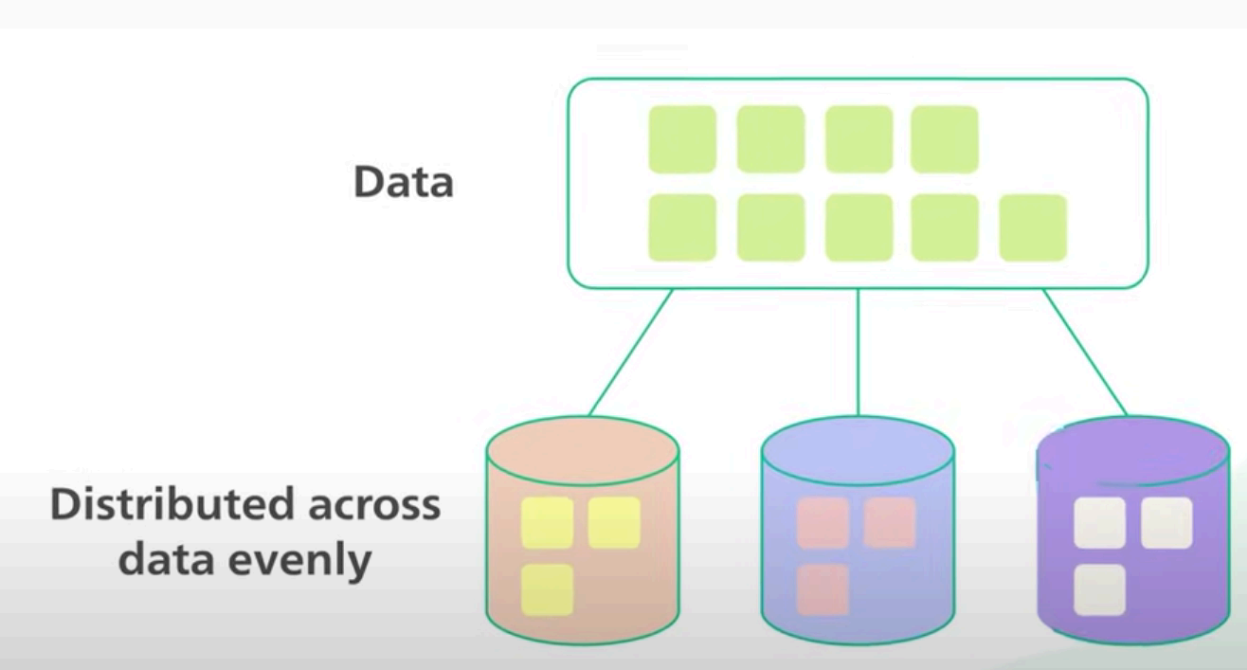
- Пока можно на словарях конкрутентых сделать
- Удалению уделить внимание, консистентность или доступность

Концепция простого хеширования

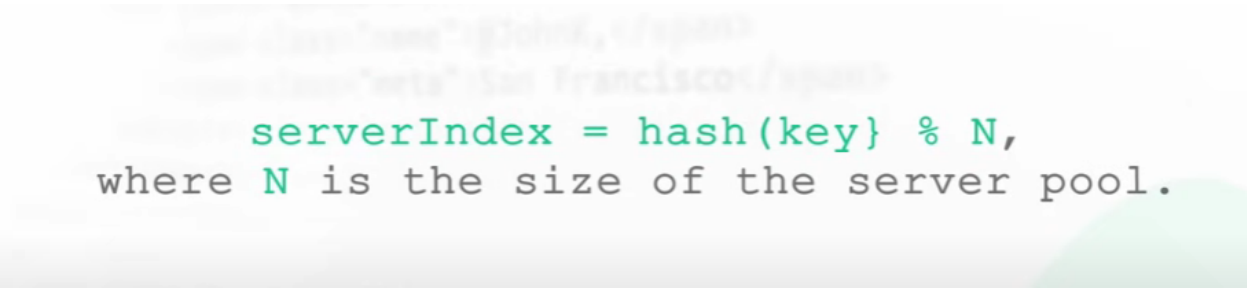
В больших системах данные располагаются на нескольких серверах



Чтобы построить такую систему с хорошей производительностью важно размещать данные равномерно между этими серверами

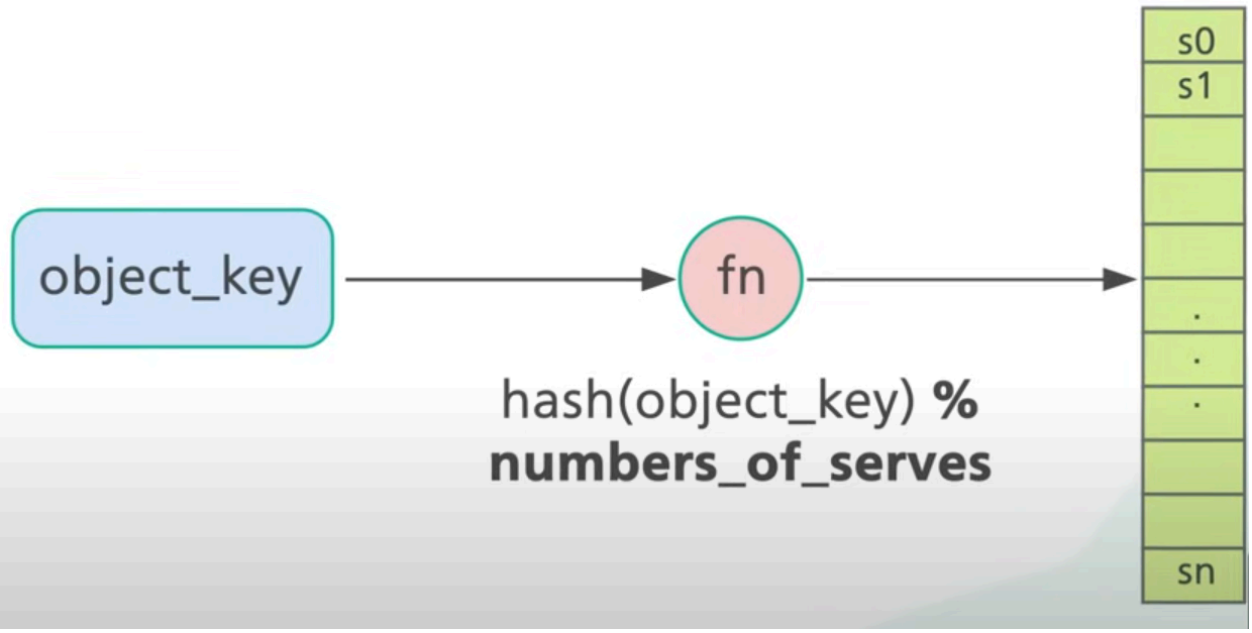


Метод распространения данных между серверами равномерно можно задать с помощью формулы

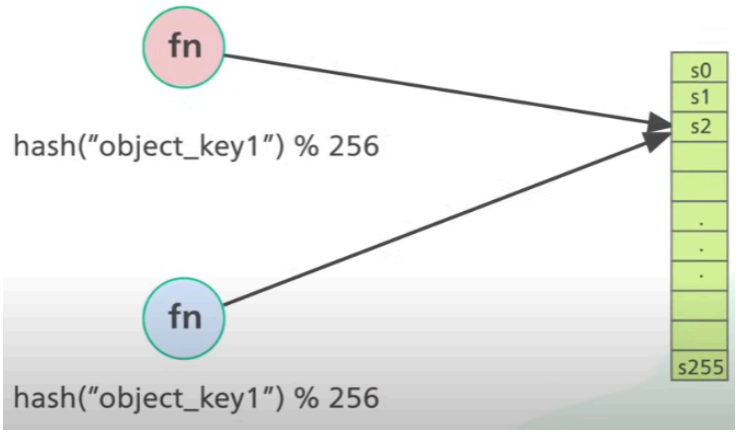


Вот как это работает: сначала для каждого объекта мы хешируем ключ с помощью функции хеширования, а это сопоставляет ключ объекта с известным диапазоном числовых значений.

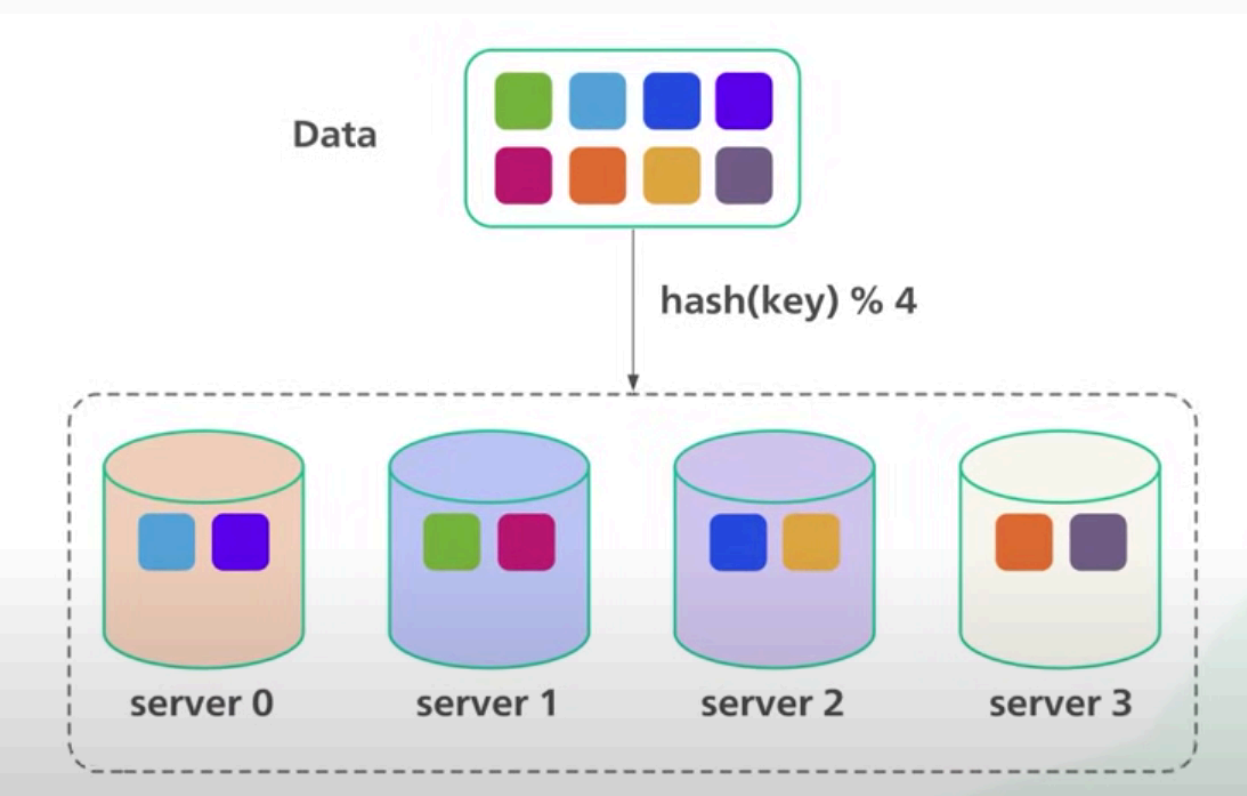
Хорошая функция хеширования равномерно распределяет хеши по всему диапазону.



Затем мы выполняем операцию деления с остатком над хешем по количеству серверов. Это определяет, к каким серверам принадлежит объект, пока количество серверов остается прежним ключ объекта всегда будет соответствовать одному и тому же серверу



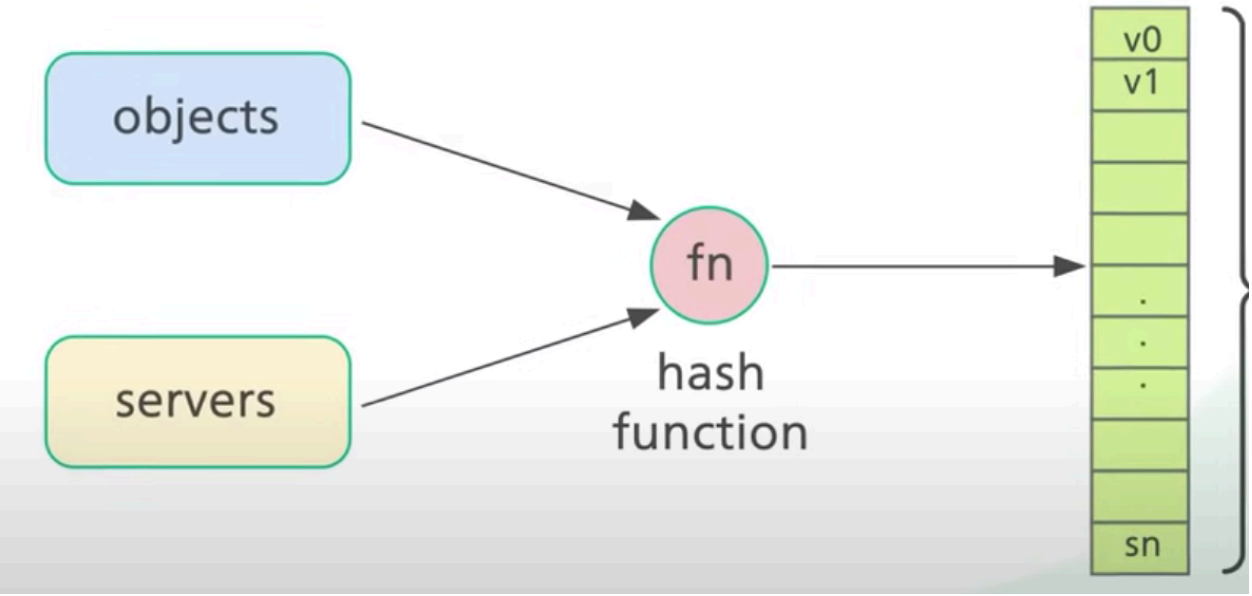
Пример для простого хеша: то есть какие-то данные распределяются между четырьмя серверами, но в настоящее время так как объем данных постоянно растет, мы не можем хранить все только на фиксированном количестве серверов, также при простом хеше если один из серверов уйдет, скорее всего, это затронет все данные. Больше данных — больше серверов.



Так, consistent hashing решают проблему, если количество серверов нужно увеличивать или от некоторых серверов нужно иногда отказываться, а в части случаев может потребоваться и для первого условия и для второго, а также если мы хотим, чтобы почти все объекты оставались назначенными на один и тот же сервер даже если количество объектов меняется

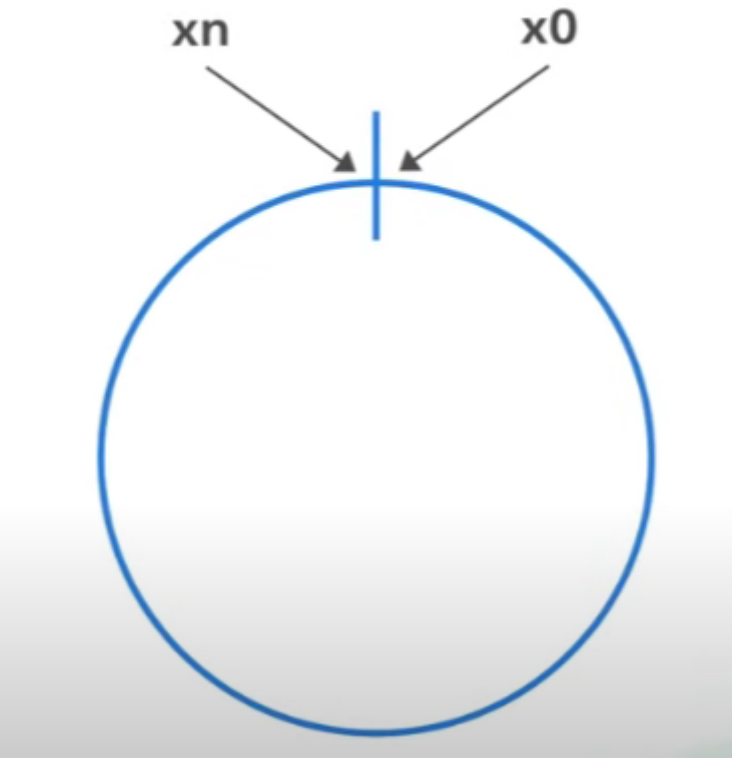
Consistent hashing

В consistent hashing мы хешируем не только ключи(данные), мы также хешируем имена серверов, они хешируются одной хеш функцией в тот же диапазон значений

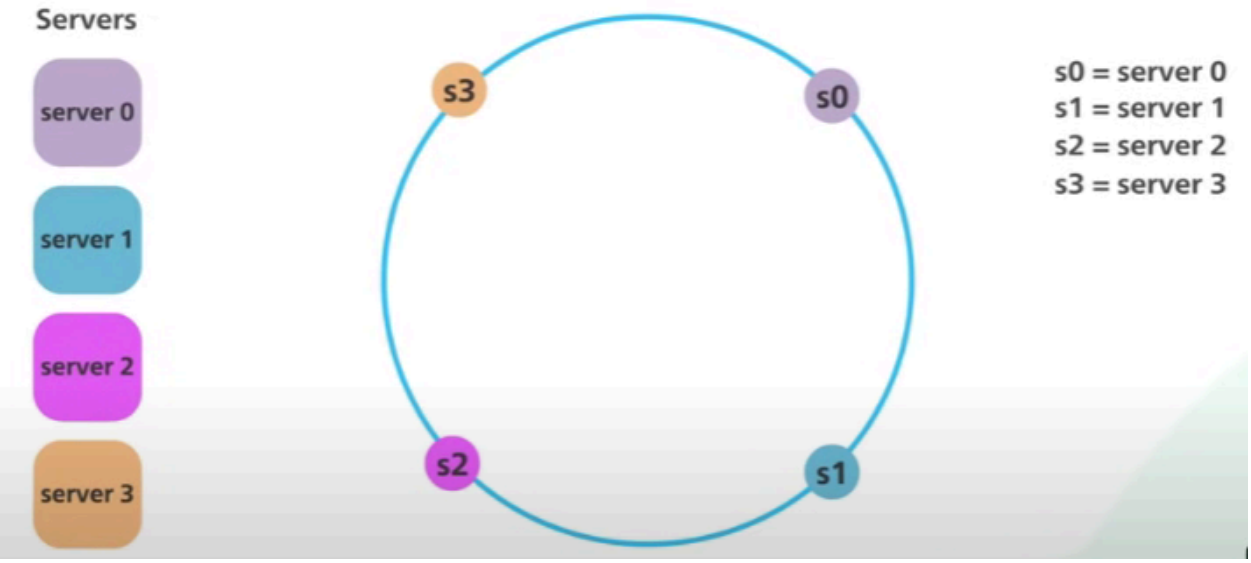


Для примера скажем, что у нас есть диапазон значений от x0 до xn. Этот диапазон называется хеш-пространством

затем мы соединяем оба конца хеш-пространства, чтобы сформировать кольцо

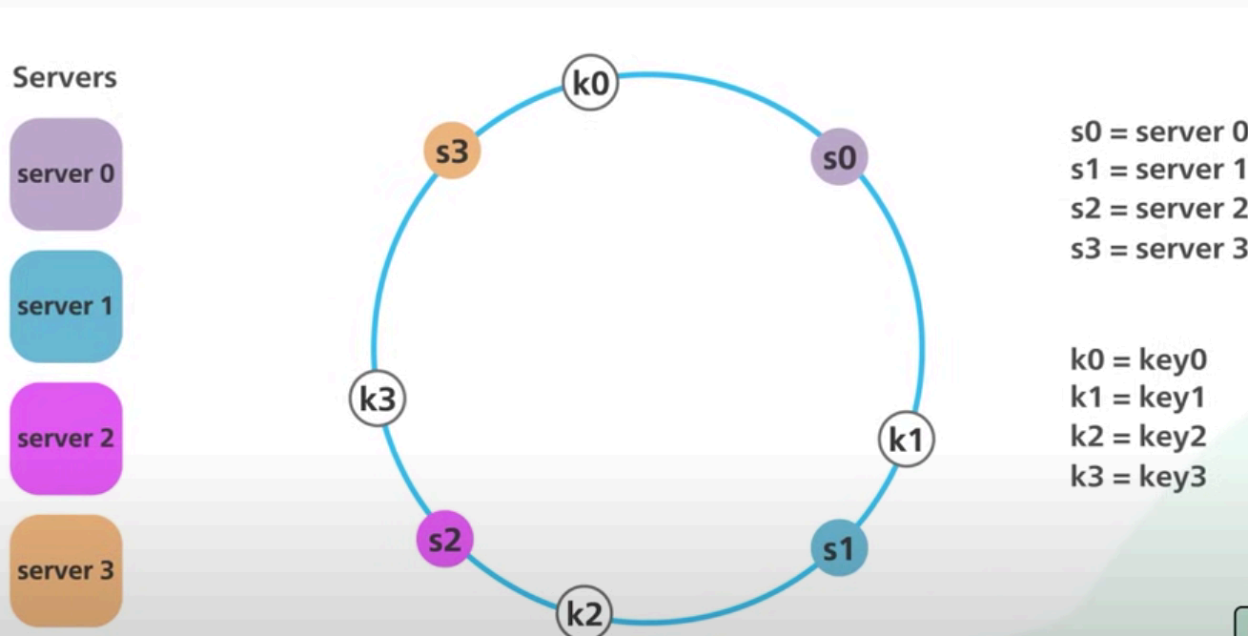


Используя функцию хеширования, мы хешируем каждый сервер по его имени или IP-адресу и помещаем сервер в кольцо

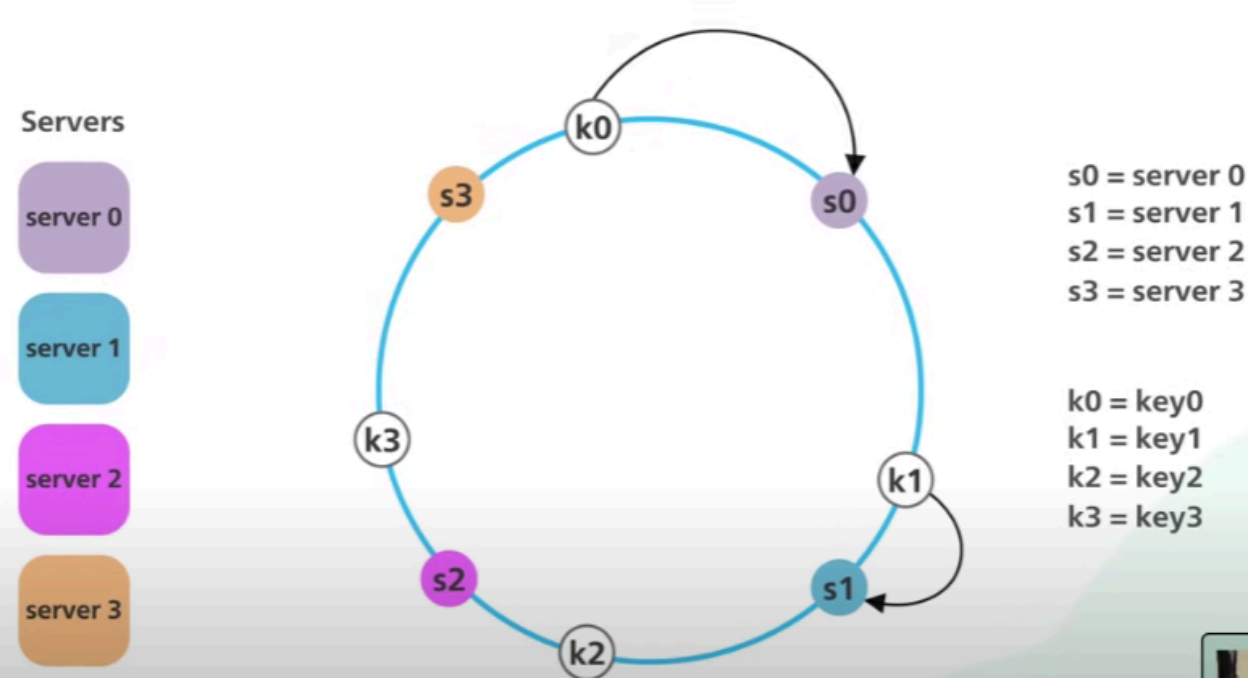


Здесь мы помещаем наши четыре сервера в кольцо. Затем мы хешируем каждый объект по его ключам с помощью той же функции хеширования, в

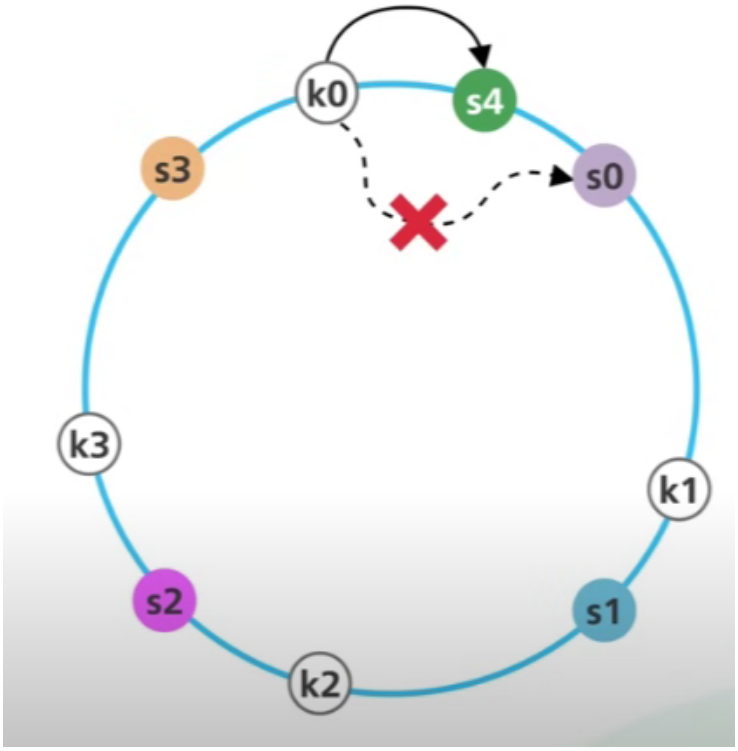
отличие от простого хеширования, где мы выполняем модульную операцию над хешем, здесь мы используем хеш напрямую для отображения ключа объекта на кольце. Вот как это будет выглядеть для наших четырех объектов



Чтобы найти сервер для определенного объекта. Мы идем по часовой стрелке от местоположения ключа объекта на кольце, пока сервер не будет найден.



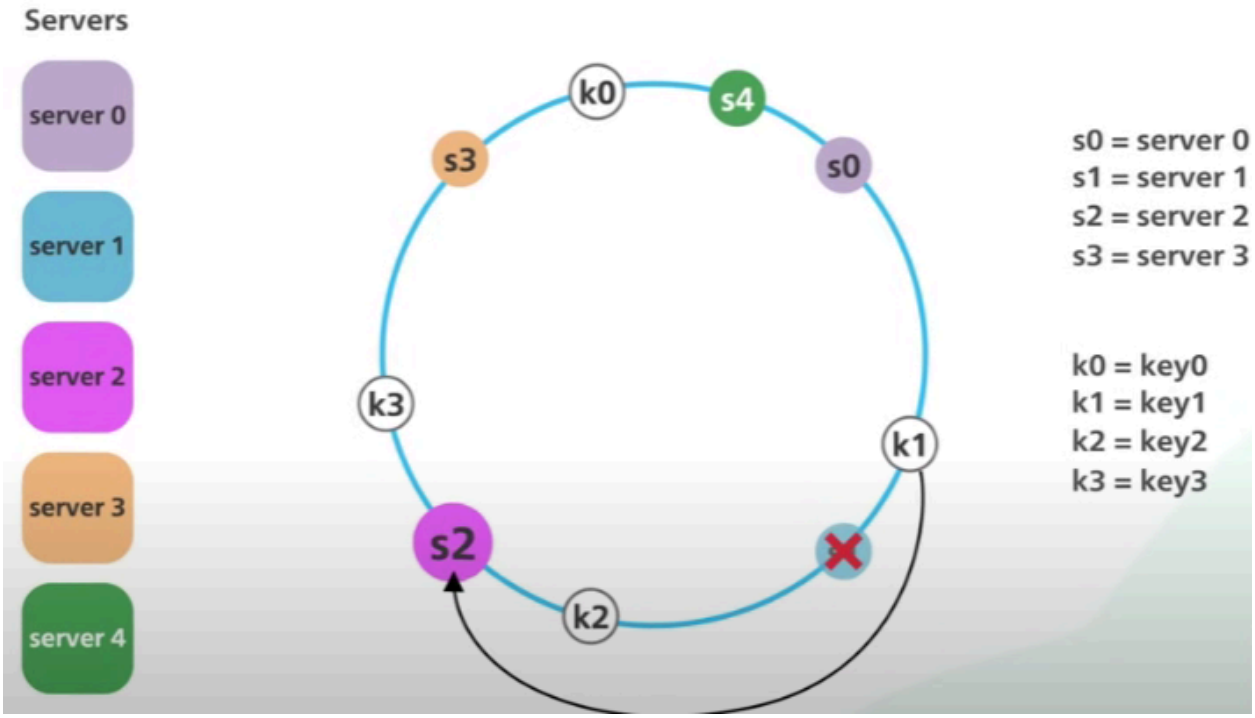
Теперь давайте попробуем добавить сервер s4 на кольцо куда-нибудь перед s0, теперь нам нужно переместить k0 на s4



Благодаря этому ключи k1, k2 и k3 никак не затронуты.

С простым хешированием при добавлении нового сервера почти все ключи необходимо переназначить, а с последовательным хешированием добавление нового сервера требует перераспределения только части ключей

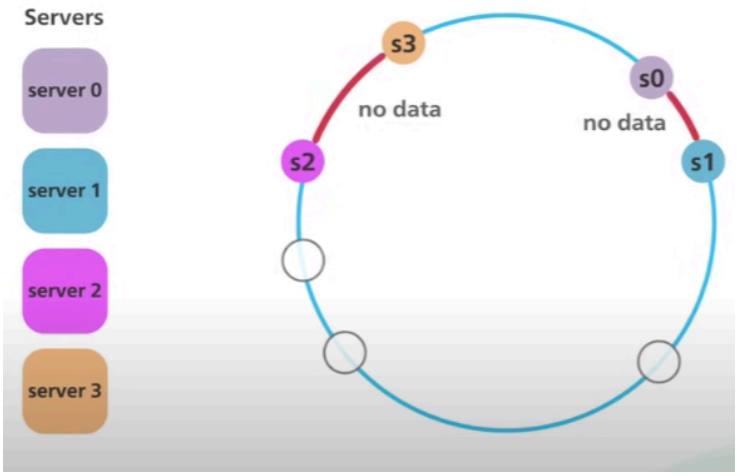
Давайте рассмотрим краткий пример удаления сервера. При удалении S1 необходимо переназначить только K1 на S2, остальные ключи не затрагиваются.



Подводя небольшой промежуточный итог тому, что мы узнали на данный момент. Во-первых, мы отображаем как серверы, так и объекты в хэш-кольце, используя равномерно распределенную хэш-функцию. Во-вторых, чтобы найти сервер для объекта, мы идем по часовой стрелке по кольцу от позиции объекта, пока сервер не будет найден.

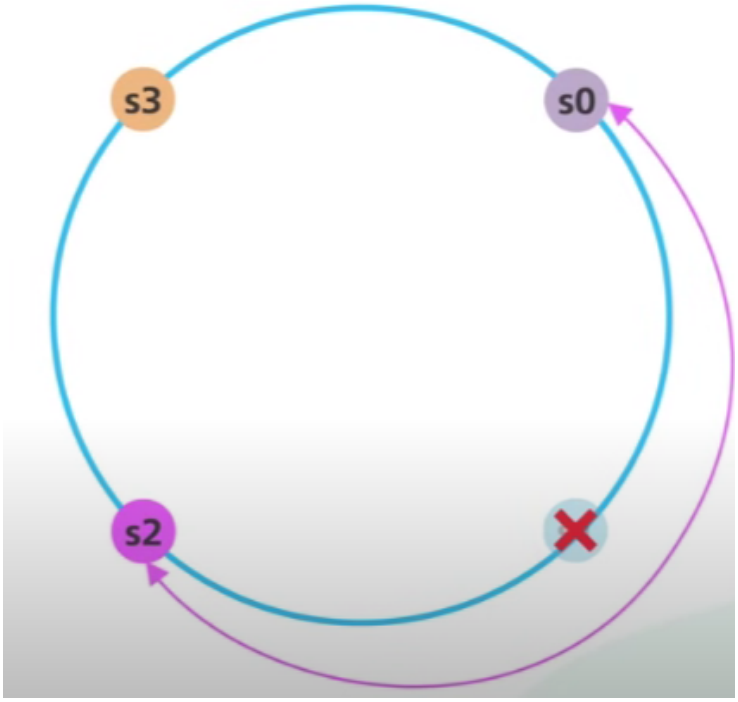
Проблемы consistent hashing

теперь давайте рассмотрим потенциальную проблему с этой конструкцией. Распределение объектов на кольце, вероятно, будет неравномерным. Концептуально мы выбираем случайные точки на кольце, и вряд ли получим идеальное разбиение кольца на сегменты одинакового размера.



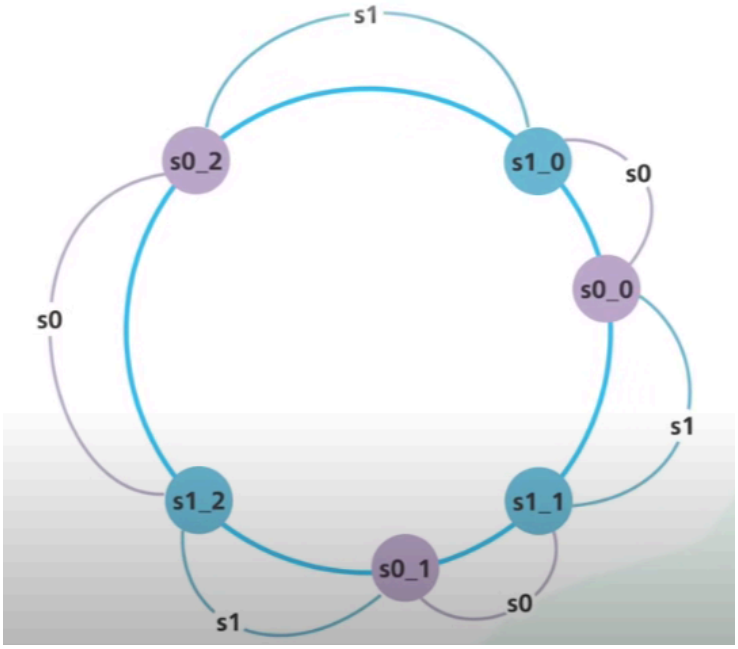
например, если серверы отображены на кольце таким образом, большинство объектов хранятся в S2, а S1 и S3 не хранят никаких данных. Эта проблема усугубляется, если серверы часто приходят и уходят

В нашем примере, даже если серверы изначально были равномерно распределены, если удалить S1, то сегмент для S2 теперь в два раза больше, чем для s0 и S3



Решение проблемы

Для решения этой проблемы используются виртуальные узлы. Идея состоит в том, чтобы каждый сервер отображался в нескольких местах на кольце. Каждое место — это виртуальный узел, представляющий сервер в этом хеш-кольце. У нас есть два сервера, каждый из которых имеет три виртуальных узла. **с виртуальными узлами каждый сервер обрабатывает несколько сегментов на кольце**



В системах реального мира количество виртуальных узлов намного больше трех, так как количество виртуальных узлов увеличивается, распределение объектов становится более сбалансированным. Наличие большего количества виртуальных узлов означает, что для хранения метаданных о виртуальных узлах требуется больше места. Это — компромисс, и мы можем настроить количество виртуальных узлов в соответствии с нашими системными требованиями

Давайте посмотрим, как согласованное хеширование используется в реальном мире некоторые популярные базы данных без SQL, такие как Amazon Dynamo DB и Apache Cassandra, используют согласованное хеширование для разделения данных, это помогает этим базам данных минимизировать перемещение данных во время повторной балансировки сети. В доставке контента используют согласованное хеширование, чтобы равномерно распределять веб-контент среди серверов. Балансировщики, такие как Google, используют согласованное хеширование, чтобы равномерно распределять постоянные соединения между внутренними серверами, это ограничивает количество соединений, которые необходимо восстановить, когда внутренний сервер выходит из строя

Как предоставить доступ к данным во время ребалансировки?

- **Репликация данных (Replication)**
  - Каждому ключу можно назначать несколько серверов (например, 3), чтобы данные дублировались.
  - Если один сервер выходит из строя, другой берет на себя его запросы.
  - Используется в Amazon DynamoDB, Apache Cassandra (eventual consistency).
- **Фоновые процессы ребалансировки (Background Rebalancing)**
  - Данные не мгновенно перемещаются между серверами, а постепенно.
  - В это время запросы могут временно перенаправляться на соседние узлы (fallback).
- **Кэширование на клиентах (Client-Side Caching)**
  - Если сервер пропал, клиент может временно использовать устаревшие данные из локального кэша.
  - Этот метод подходит для сценариев, где допустима eventual consistency.