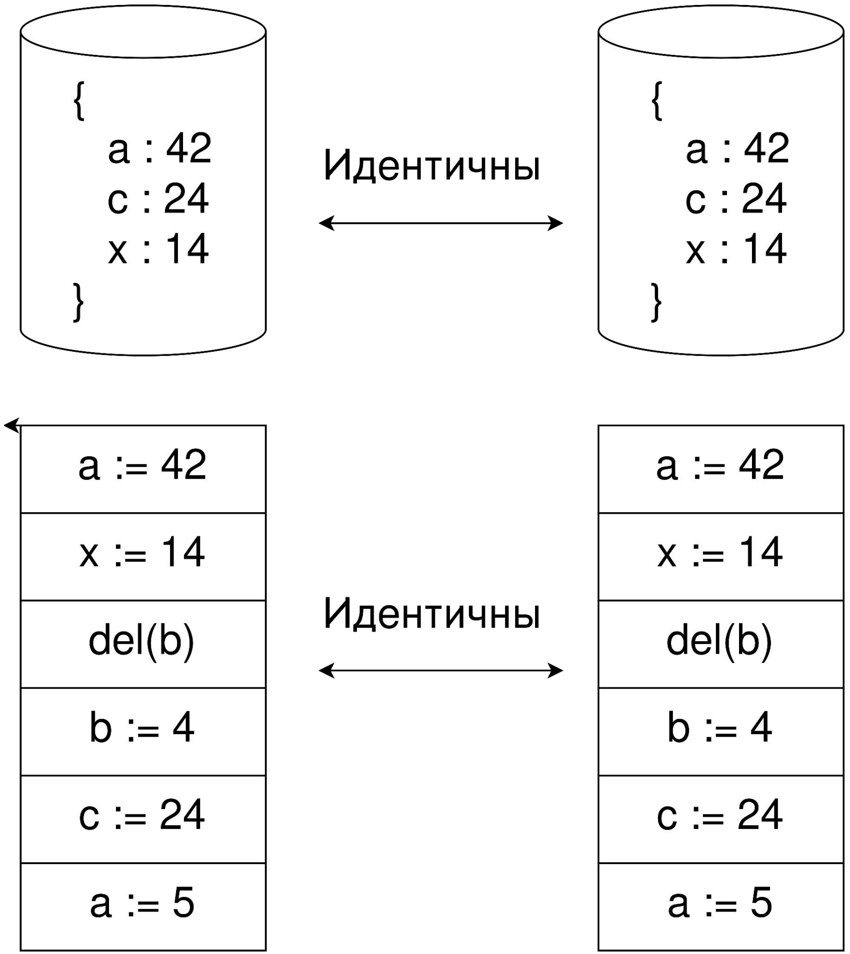
Шардирование

Илья Кокорин
kokorin.ilya.1998@gmail.com

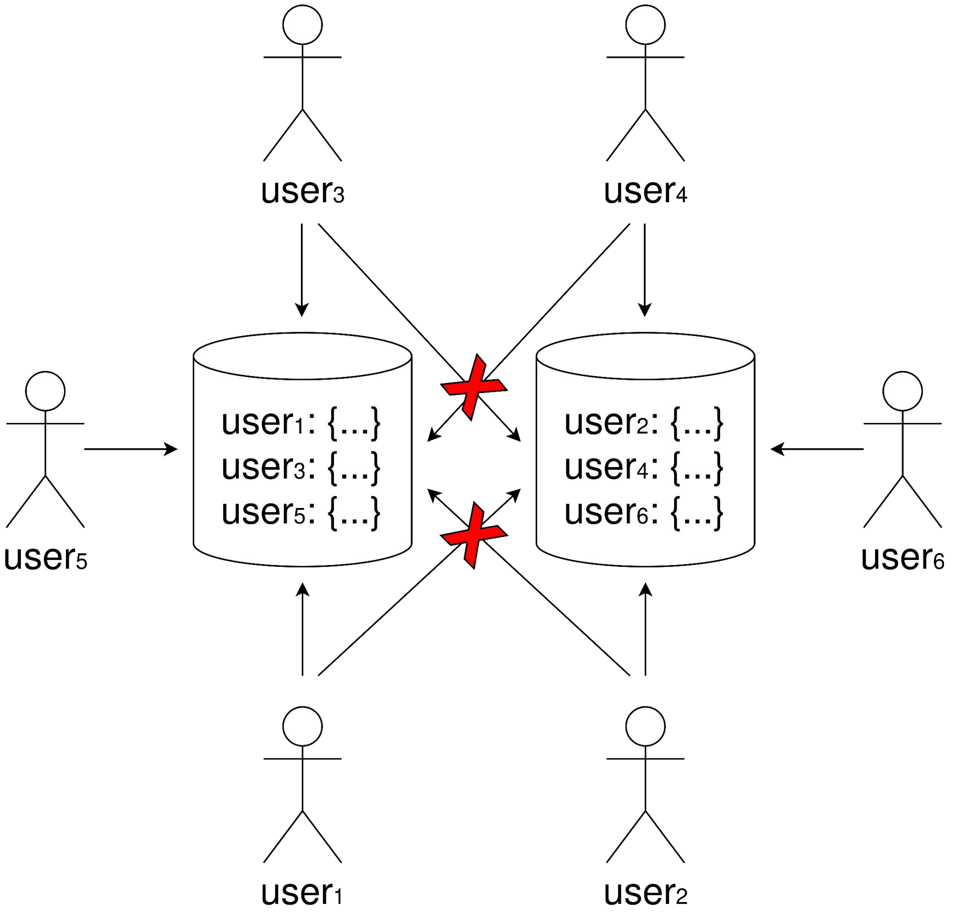
В предыдущих сериях

- Replicated State Machines (RSM)
  - Все узлы системы хранят копию одних и тех же данных
  - Синхронизируем реплики с помощью консенсуса
- Но зачем?



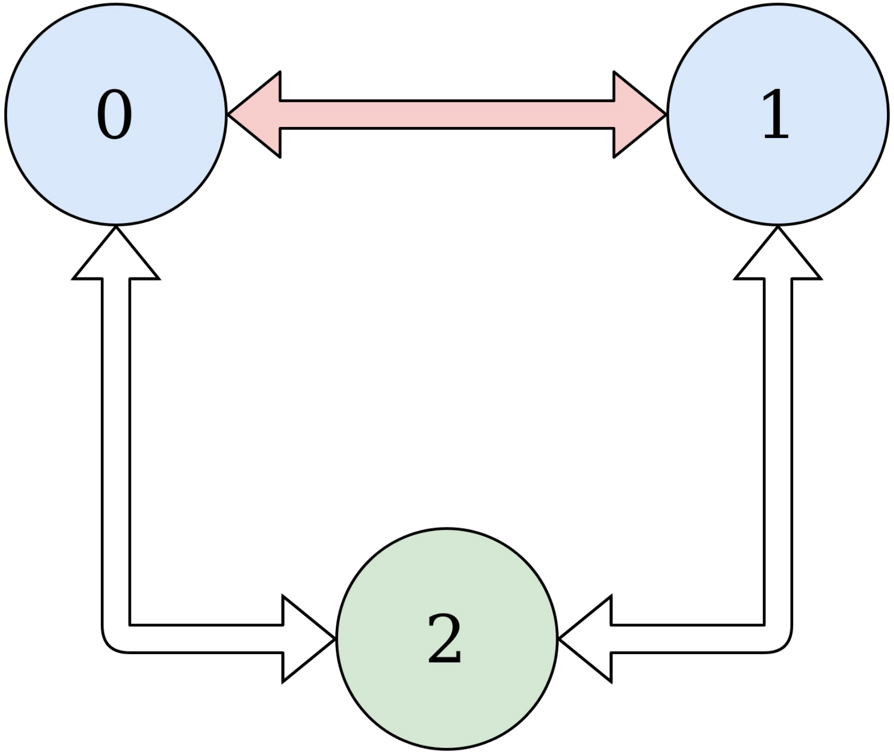
Шардирование: мотивация

- Храним данных больше, чем доступно памяти на одном узле
- Каждый узел обслуживает запросы только к тем данным, которые он хранит
- Не обслуживает запросы к чужим данным
- Больше запросов в секунду



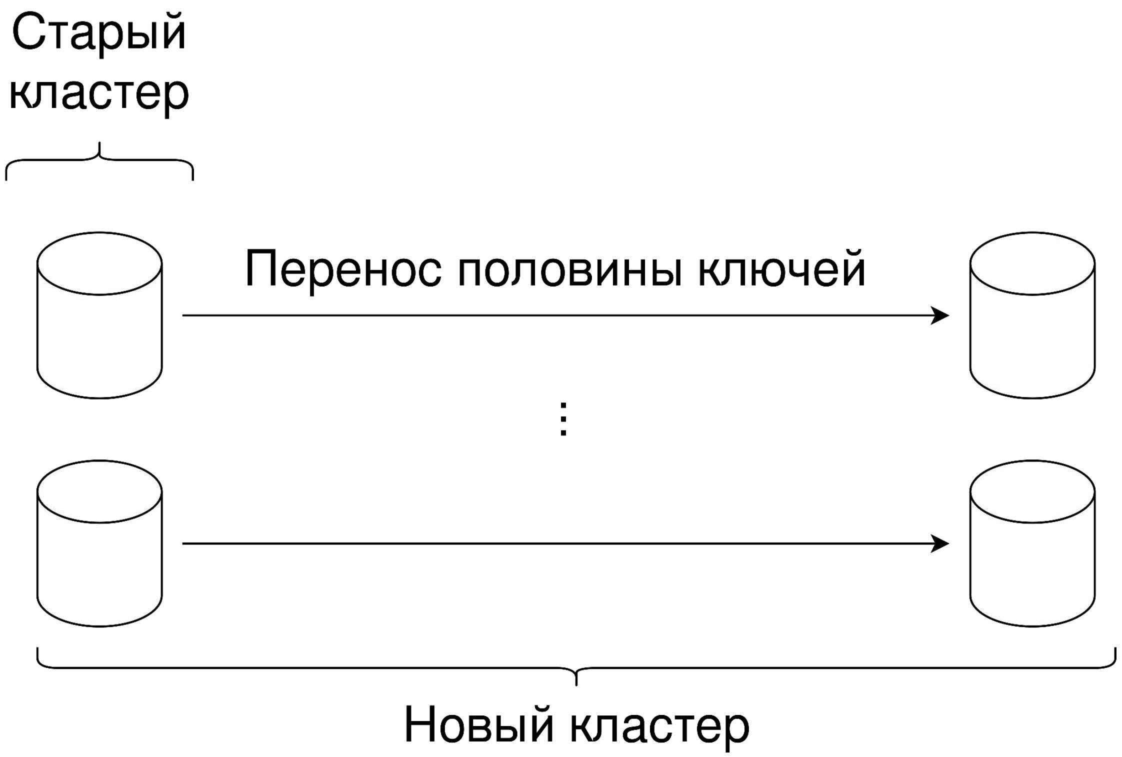
Шардирование по модулю

- `node_id := hash(key) % p`
- Узлы заполнены равномерно
- При добавлении/удалении узла приходится перемещать Θ(N) ключей
  - 15 % 2 = 1
  - 15 % 3 = 0
- Перемещаем ключи между существующими узлами



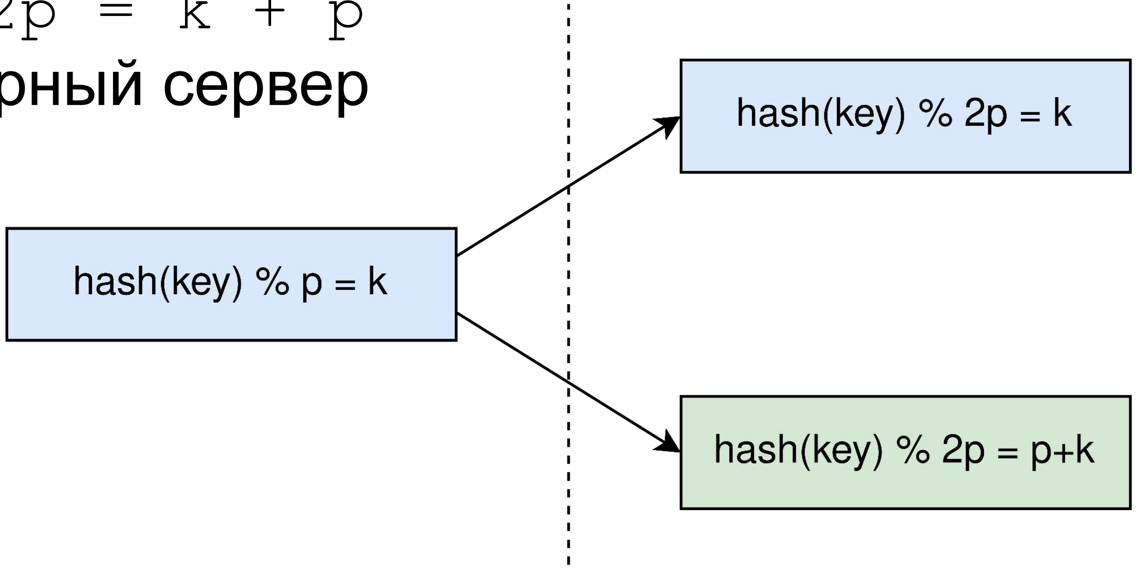
Шардирование по модулю: расширение

- Можем расширить кластер в два раза
- Для каждого сервера покупаем ещё один
- Переносим половину ключей на новый сервер



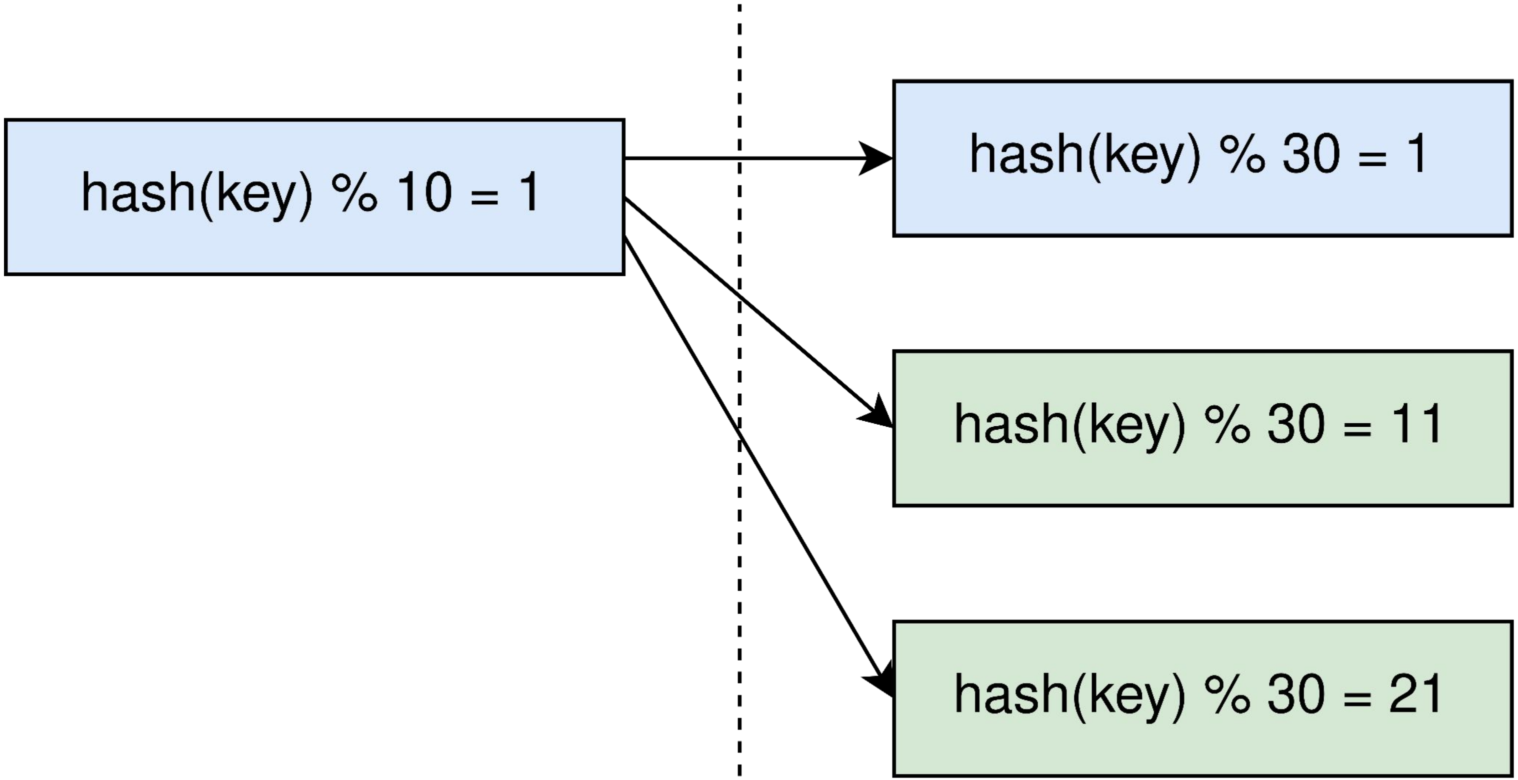
Шардирование по модулю: расширение

- Пусть `hash(key) % p = k`
  - Рассмотрим k-ый сервер
- Значит, `hash(key) % 2p = k`
  - Эти ключи оставляем на k-ом сервере
- Или `hash(key) % 2p = k + p`
  - Переносим на парный сервер
- Можно удалить половину кластера обратной операцией



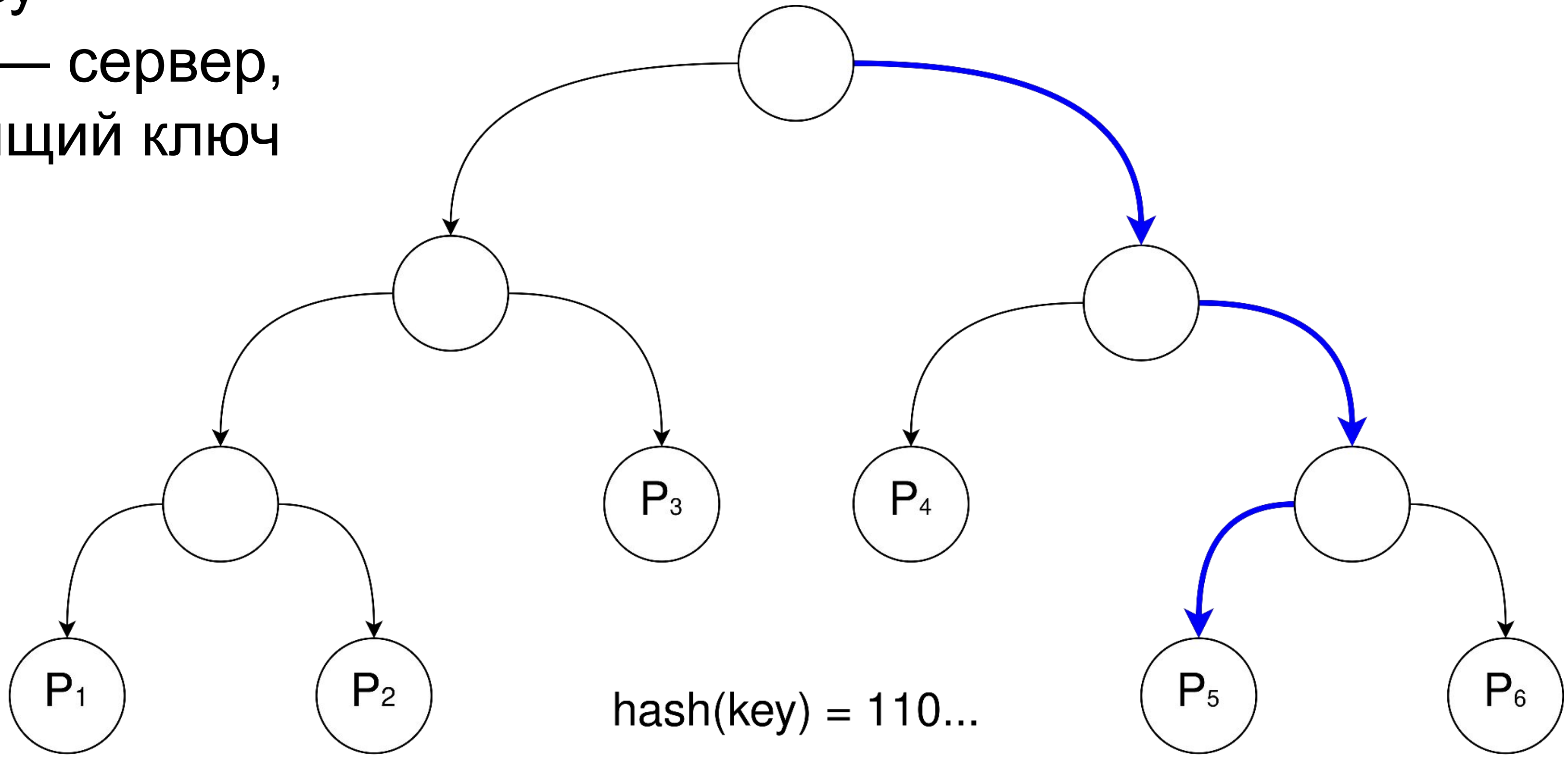
Шардирование по модулю: расширение

- Можно расширять в любое натуральное число раз



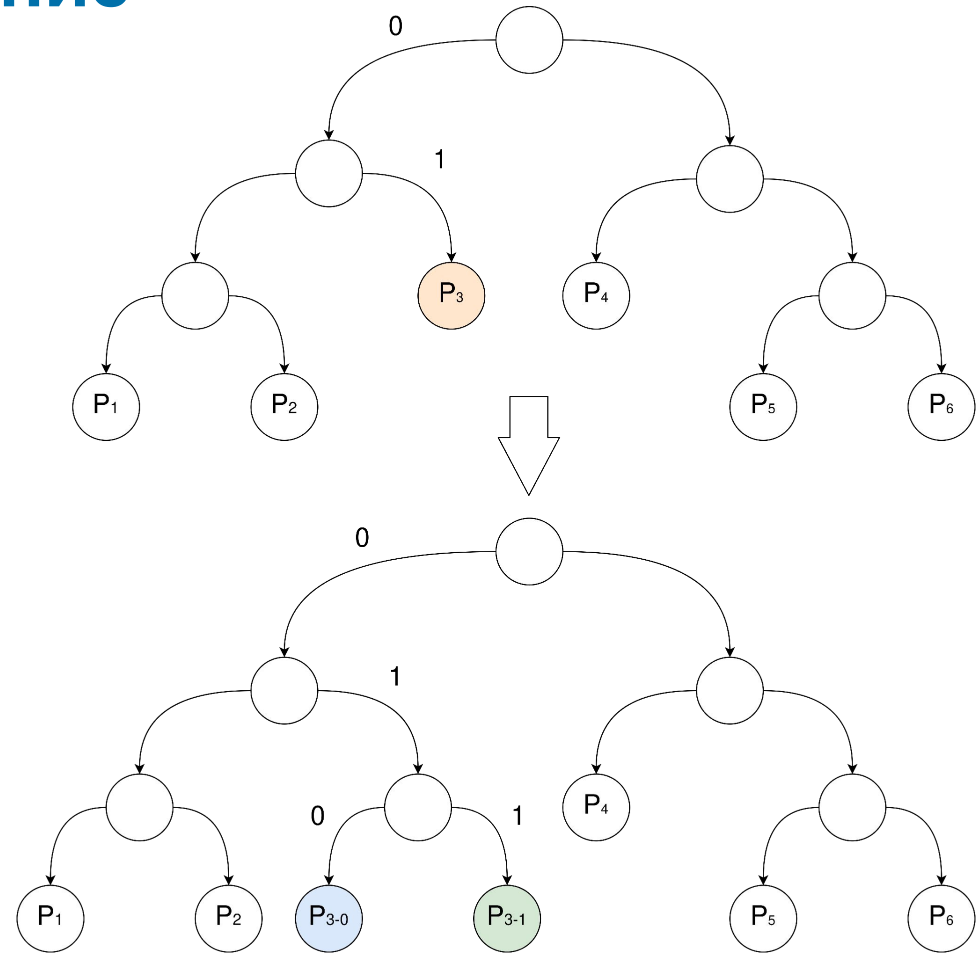
Битовый бор

- Берём битовое представление хеша и спускаемся по дереву
- Лист — сервер, хранящий ключ



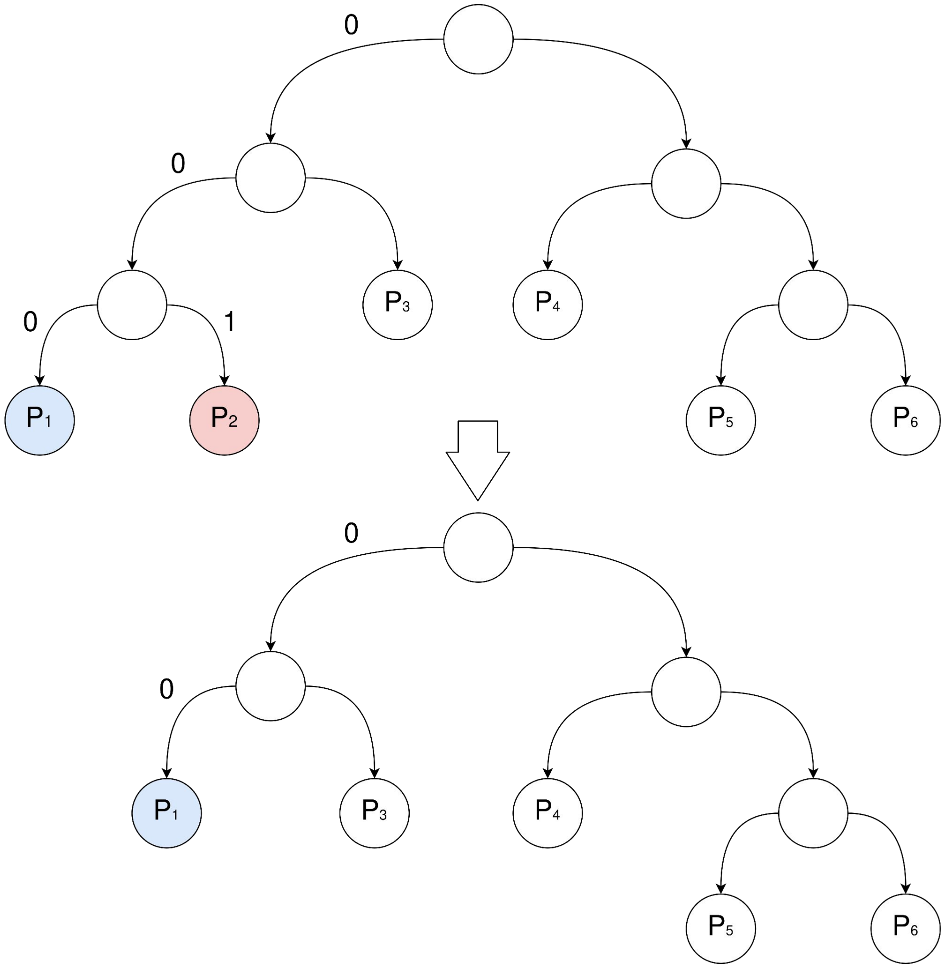
Битовый бор: расширение

- Расщепляем вершину на две
- Переносим примерно половину ключей на новый сервер
- Определяем по i-ому биту хеша
- Из одного заполненного сервера получаем два заполненных наполовину



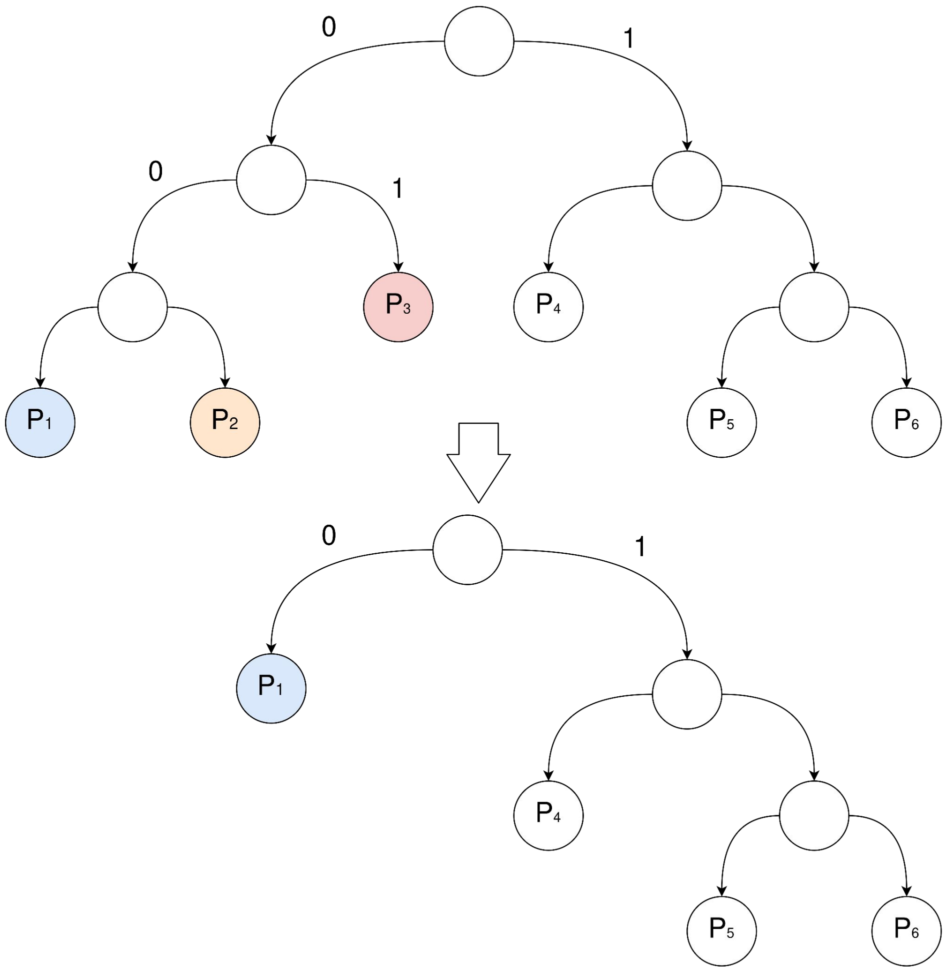
Битовый бор: удаление

- Если у удаляемой вершины есть брат-лист, переносим все ключи на него
- Из двух наполовину заполненных серверов получаем один заполненный целиком



Битовый бор: удаление

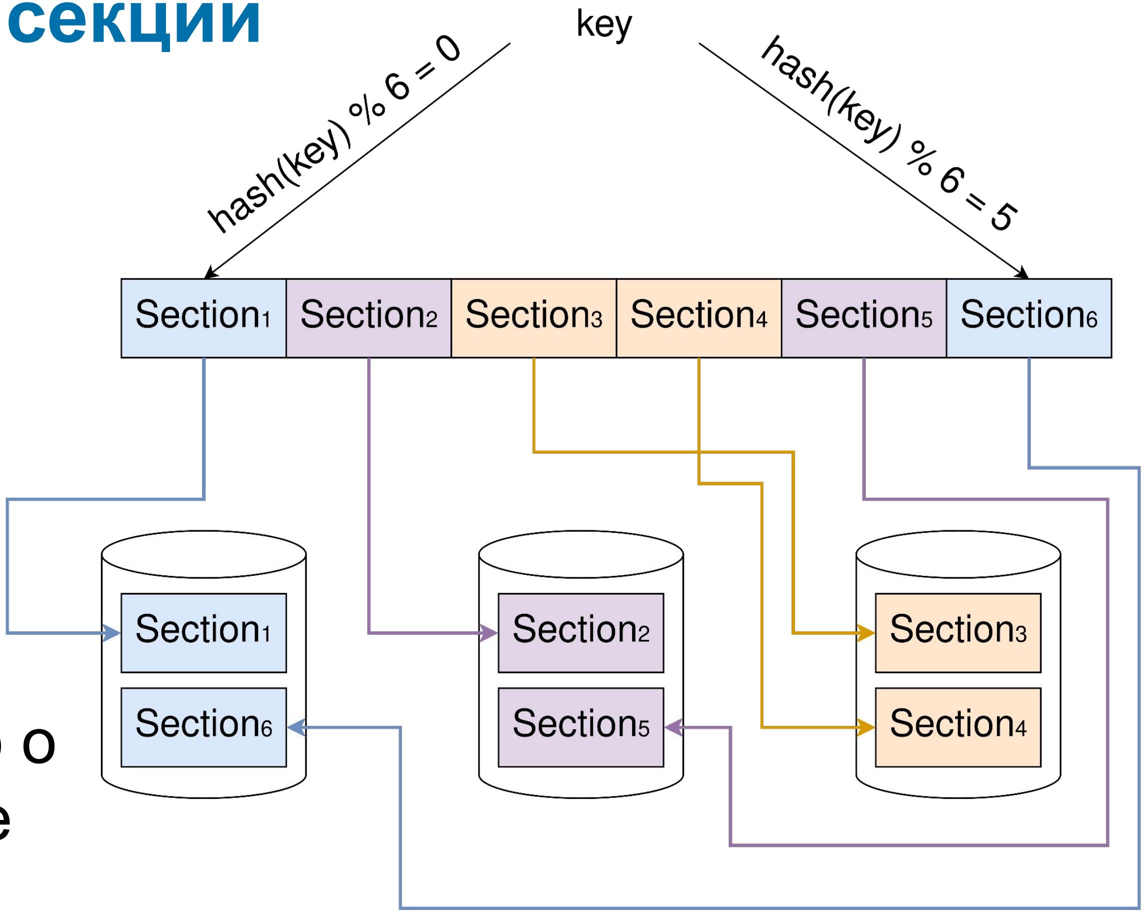
- Если у удаляемой вершины нет брата-листа, можем заменить только целое родительское поддерево на один лист
- Приходится удалять и другие листья



Постоянное число секций

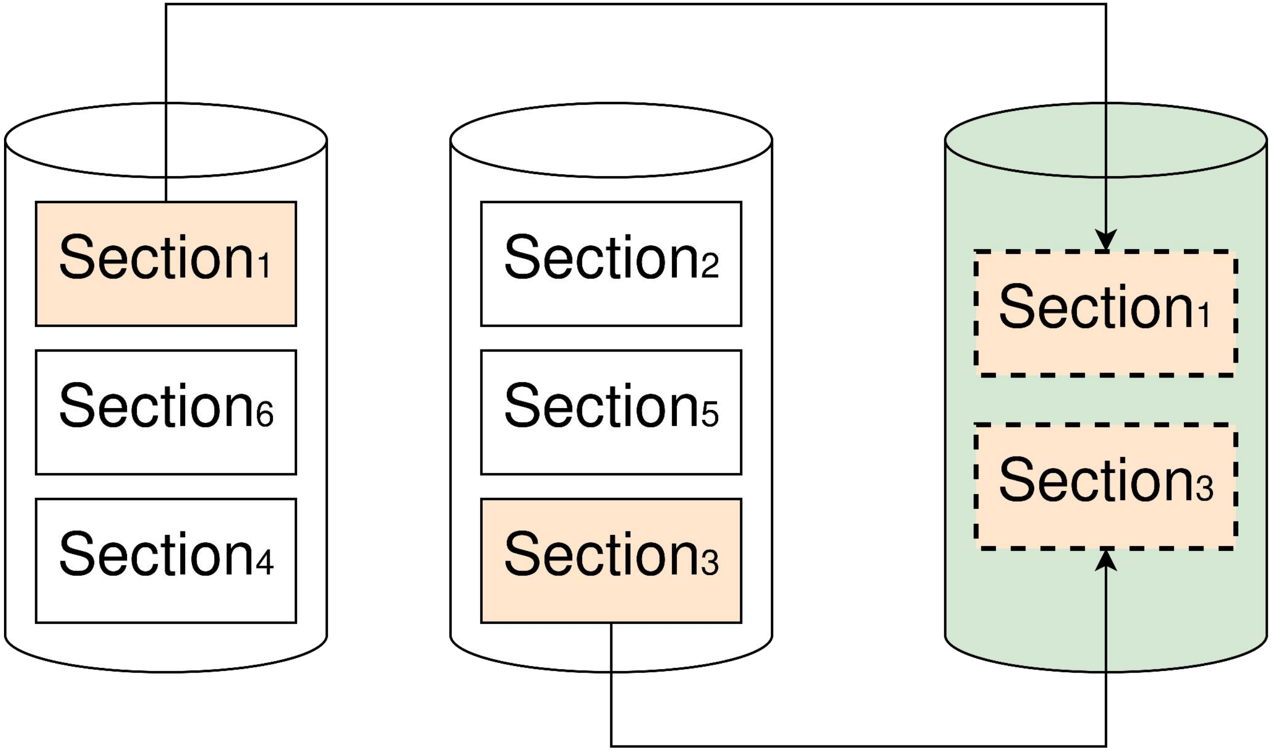
- Заводим N секций
- Ключи по секциям распределяем тривиально
- Секции храним на серверах

- Централизованно храним информацию о том, какая секция где хранится



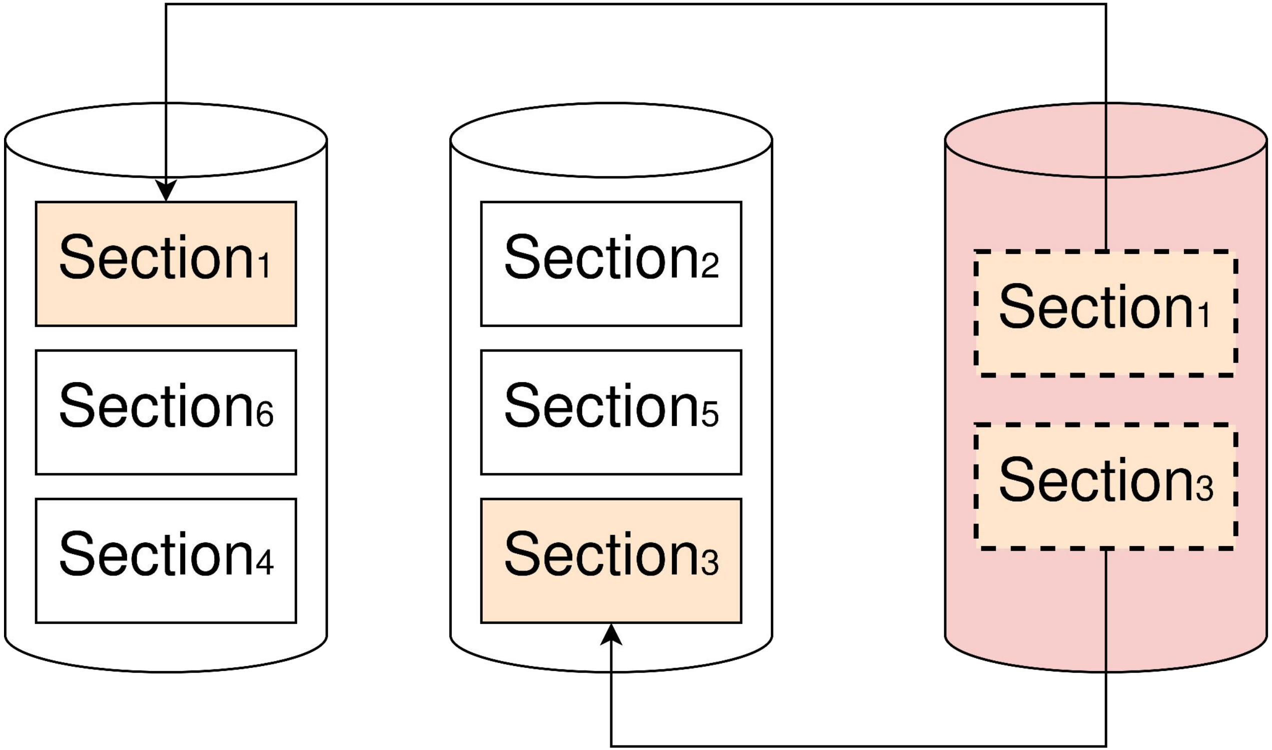
Добавление сервера

- Переносим часть секций на новый сервер



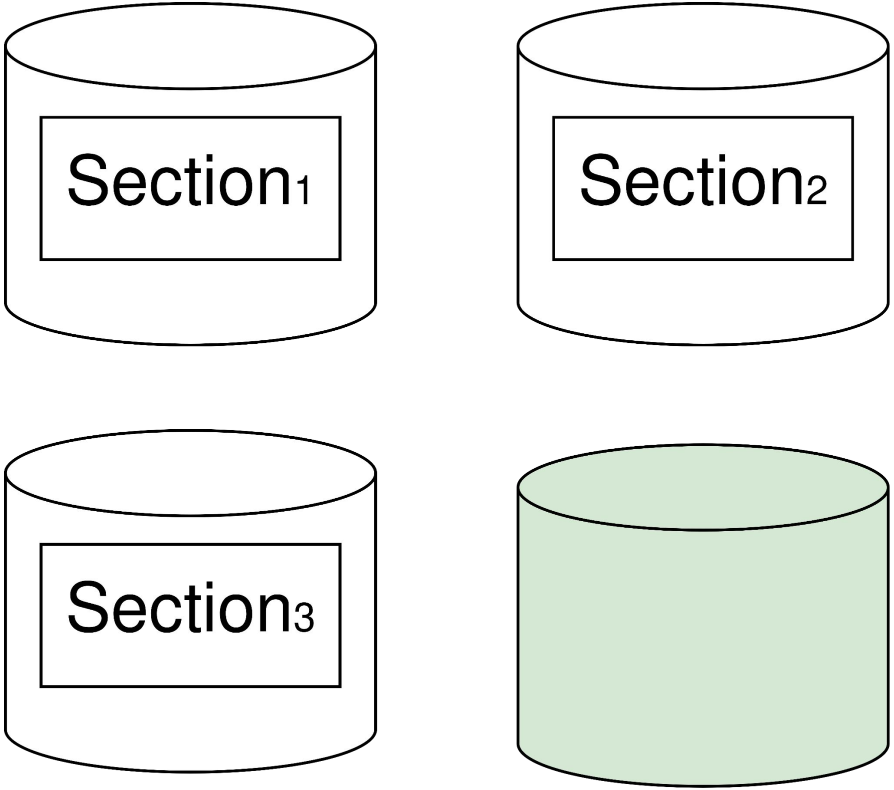
Удаление сервера

- Переносим секции с удаляемого сервера на оставшиеся



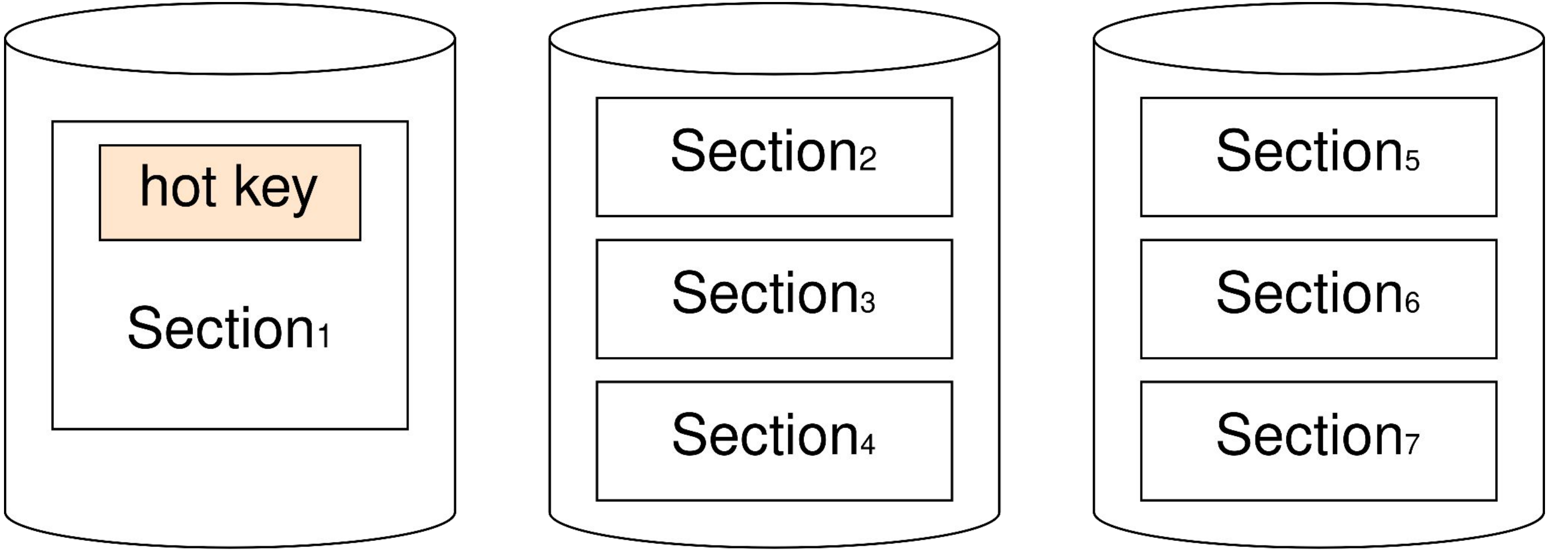
Выбор числа секций

- Нельзя слишком мало
- Иначе на очередному серверу не достанется секции



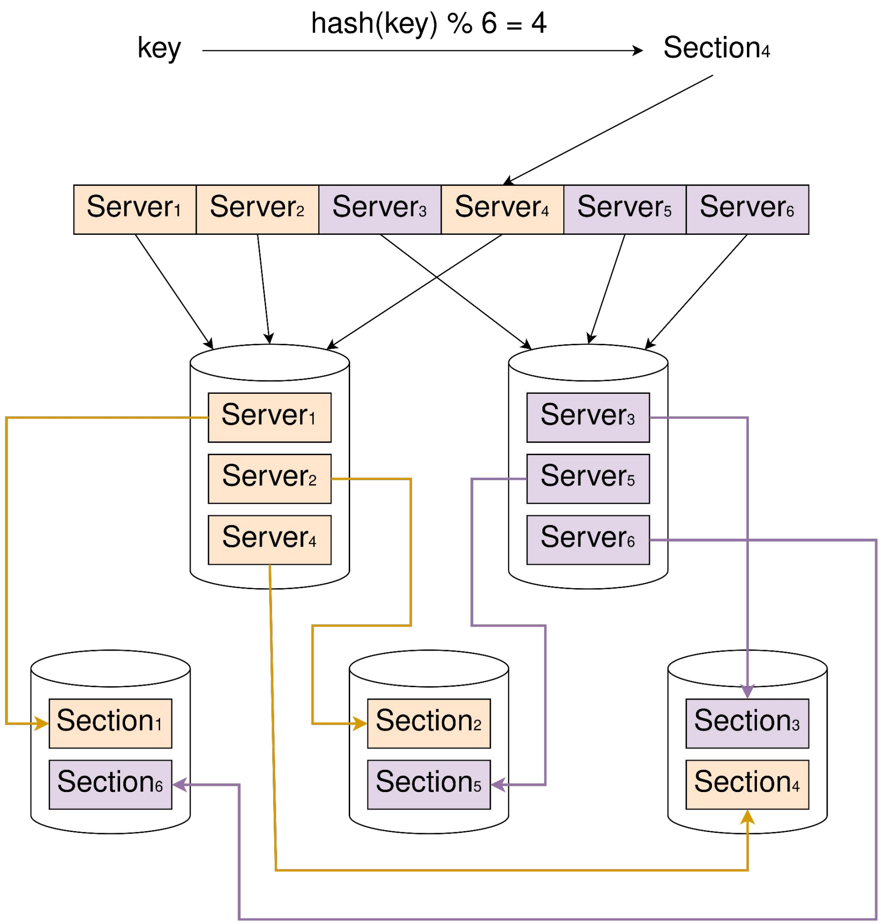
Выбор числа секций

- Чем больше секций, тем больше возможностей для балансировки
- Меньше секций на нагруженные сервера, больше на разгруженные



Многоуровневое шардирование

- Нужно хранить информацию вида `section → server`
- Её тоже можно шардировать
- Её меньше, чем исходной информации
- Для хранения нужно меньше серверов

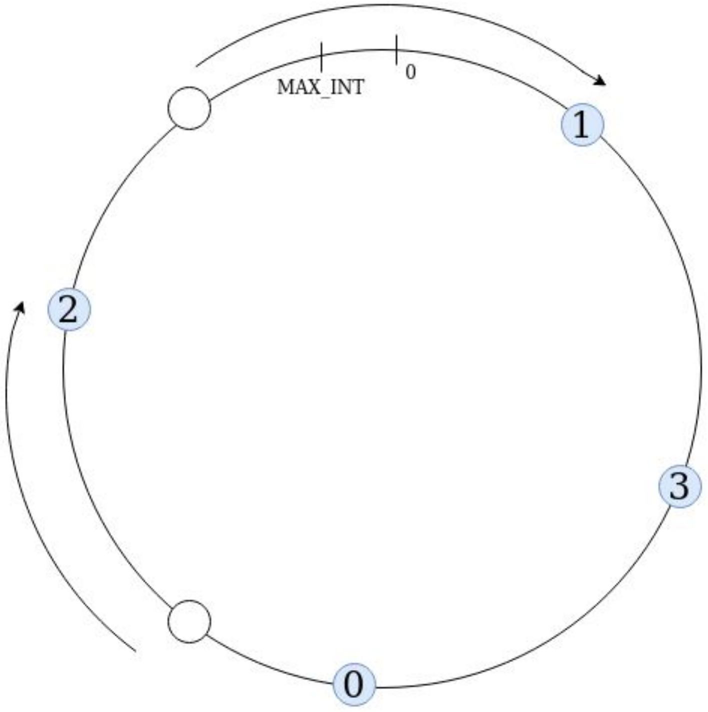


Rendezvous hashing

- `node_id = argmax hash(key || i)`
  `i= 1 to P`
- Равномерное распределение ключей
  - Если хеш-функция “хорошая”
- При добавлении узла каждый существующий узел перемещает только те ключи, которые должны перейти на новый узел
  - `hash(key || new_id) > hash(key || cur_id)`
- При удалении узла перемещаются только те ключи, которые располагались на этом узле
- Поиск узла по ключу за O(K)

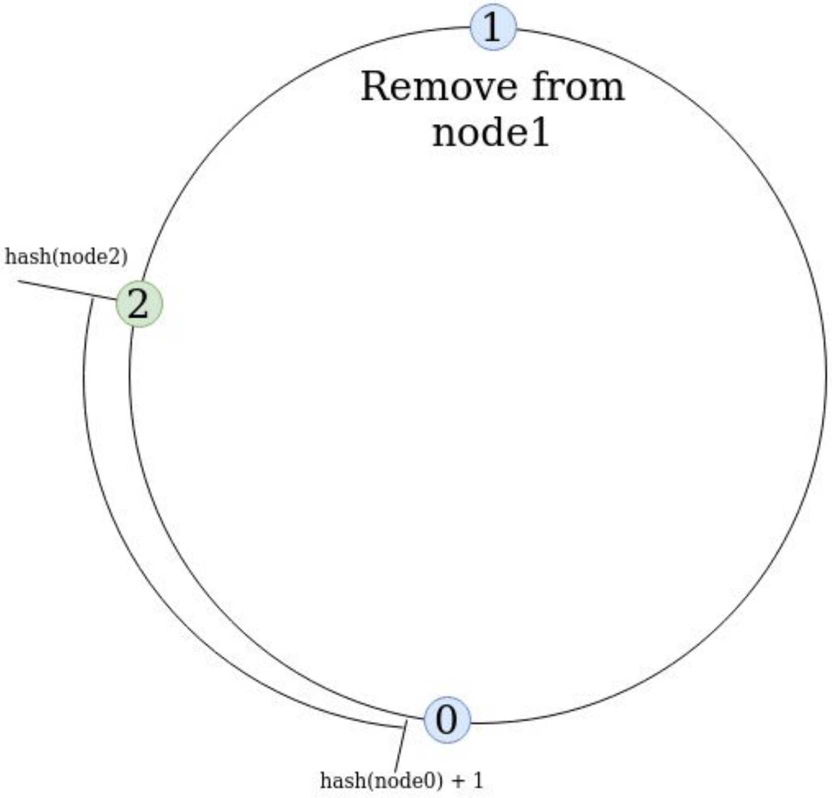
Консистентное хеширование

- Каждому узлу соответствует точка на круге
  - hash(node_name)
- Проецируем ключ на круг
  - hash(key)
- Ищем следующую по часовой стрелке точку, соответствующую узлу



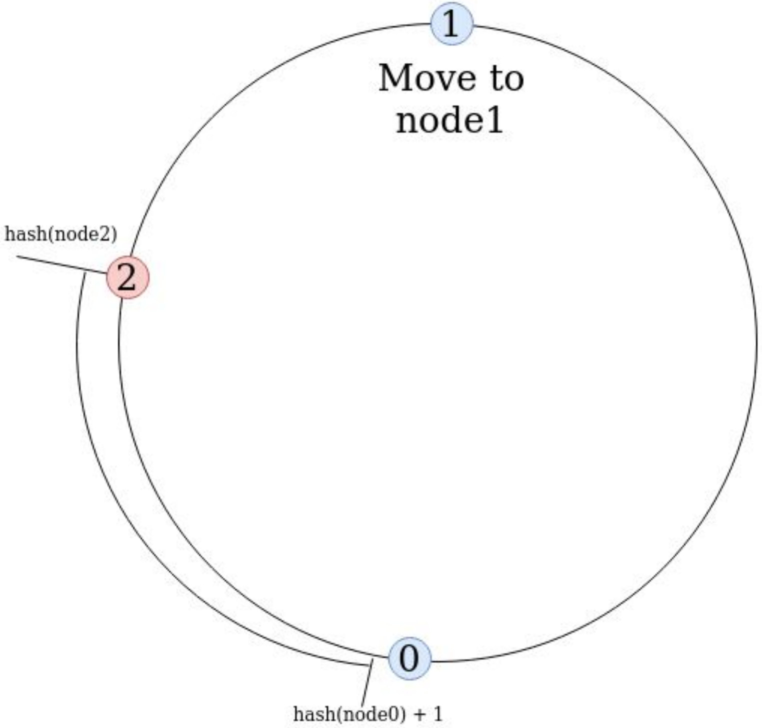
Консистентное хеширование: добавление узла

- Проецируем новый узел на круг
- Ищем предыдущий и следующий по кругу узлы
- Перемещаем часть ключей со следующего по кругу узла на новый узел
- Ключи перемещаются только на новый узел



Консистентное хеширование: удаление узла

- Аналогично добавлению
- Ключи перемещаются только с удаляемого узла

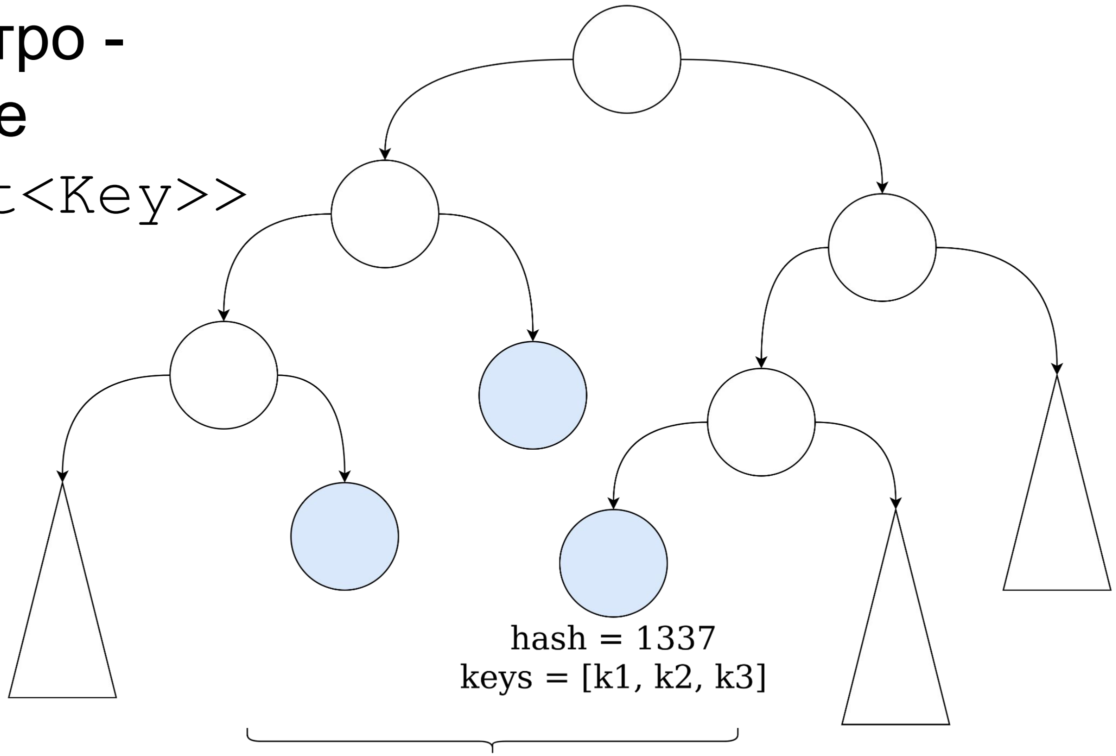


Хранение списка узлов

- В дереве поиска
  - O(log K) на поиск узла по ключу
  - O(log K) на добавление/удаление узла
- В сортированном массиве
  - O(log K) на поиск узла по ключу
  - O(K) на добавление/удаление узла

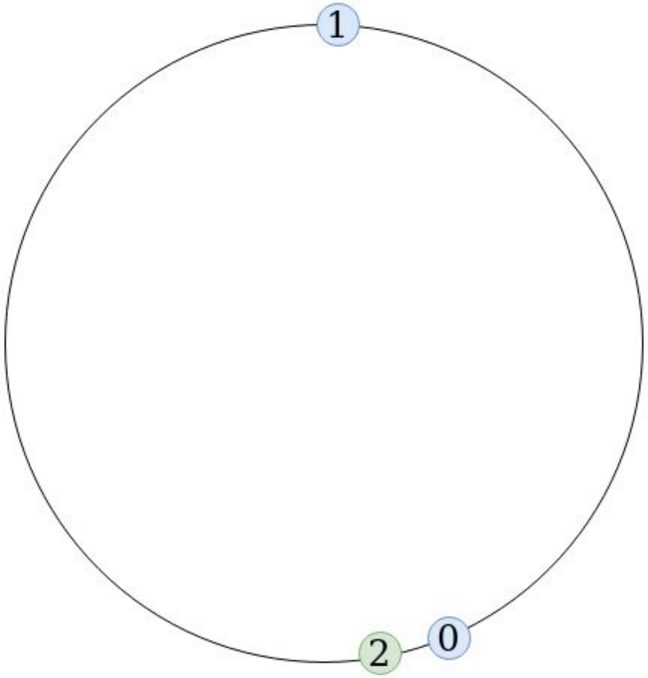
Перемещение ключей

- Перемещаются только непрерывные отрезки хешей вида `[hash(prev_node) + 1; hash(cur_node)]`
- Чтобы делать это быстро - храним на каждом узле `TreeMap<Hash, List<Key>>`



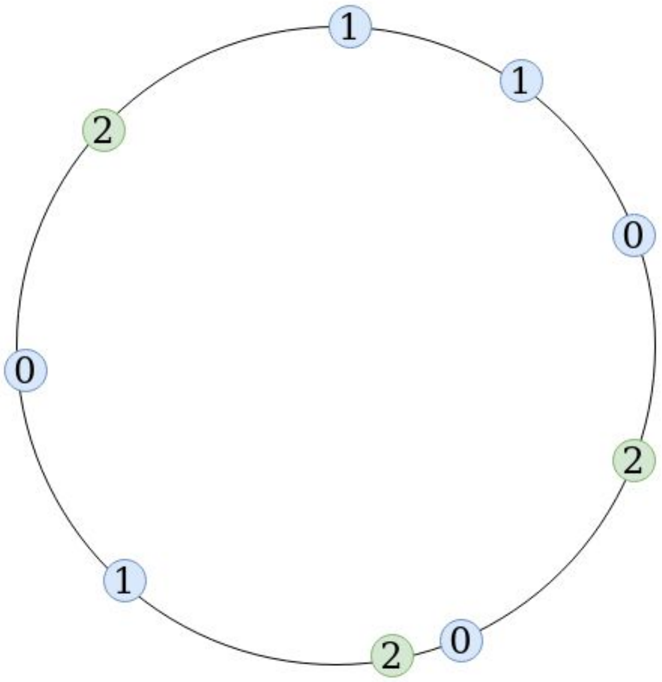
Неравномерное распределение ключей

- Такое возможно, так как отображения узлов на круг выбираются случайно
- При добавлении нового узла все свои ключи он забирает у единственного существующего узла
- При удалении узла мы перемещаем все его ключи на единственный узел



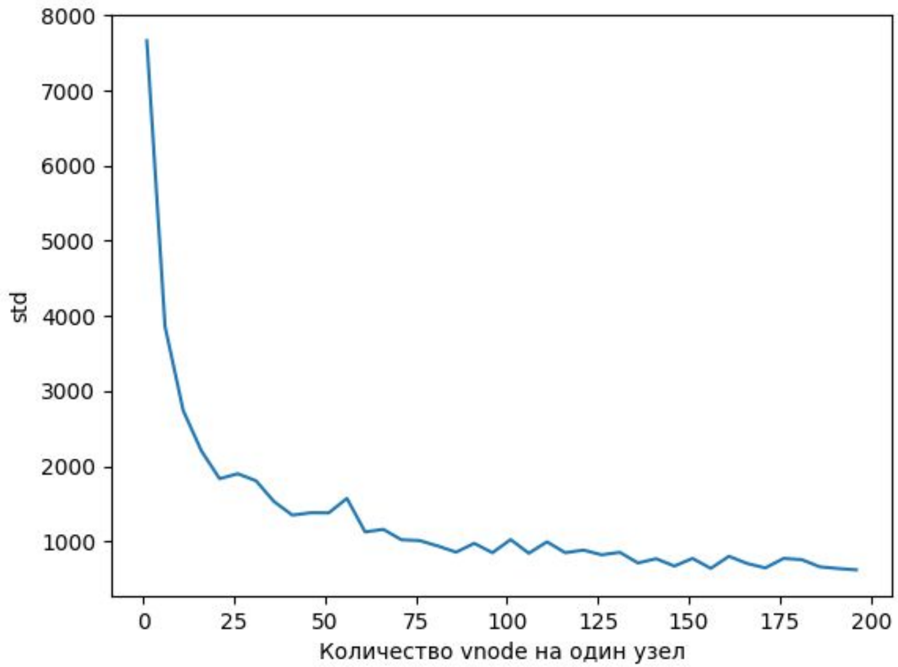
Консистентное хеширование: vnodes

- Каждому физическому узлу соответствует несколько виртуальных
  - `hash(node_name || 0), hash(node_name || 1), ...`
  - Чем больше - тем равномернее распределение
  - И больше нагрузка на память и время
- Учитываем “вес” узла: чем больше vnodes, тем больше ключей



vnodes и равномерность распределение

- Фиксируем количество узлов
- Меняем количество vnode
- Генерируем случайные ключи
- Для каждого узла считаем, сколько ключей попало на этот узел



- Считаем стандартное отклонение (чем меньше - тем лучше) с усреднением по нескольким запускам

JumpHash

- Пусть `ch(key, n)` — индекс узла, хранящего ключ `key`
  - `0 ≤ ch(key, n) < n`
- `ch(key, 1) = 0` для всех ключей
- Переход от K серверов к K + 1:
  - Каждый ключ с вероятностью $\frac{1}{K+1}$ переходит на сервер с индексом K (новый узел)
- Сервера не удаляются, только добавляются новые
  - Количество данных в системе только растёт
  - При выходе сервера из строя мы не удаляем его из кластера, а заменяем его другим сервером с тем же индексом и теми же данными

JumpHash: доказательство равномерности распределения

$0 \le \xi_n < n$

Докажем, что $P\left(\xi_n = i\right)\big|_{0 \le i < n} = \frac{1}{n}$

База: $P(\xi_1 = 0) = 1$

Переход: покажем, что

$P\left(\xi_n = i\right)\big|_{0 \le i < n} = \frac{1}{n} \Rightarrow P\left(\xi_{n+1} = i\right)\big|_{0 \le i < n+1} = \frac{1}{n+1}$

- $i = n : P\left(\xi_{n+1} = n\right) = \sum\limits_{i=0}^{n-1} P\left(\xi_n = i\right) \cdot \frac{1}{n+1} = \sum\limits_{i=0}^{n-1} \frac{1}{n} \cdot \frac{1}{n+1} = \frac{1}{n+1}$
- $i \ne n : P\left(\xi_{n+1} = i\right) = P\left(\xi_n = i\right) \cdot \left(1 - \frac{1}{n+1}\right) = \frac{1}{n} \cdot \frac{n}{n+1} = \frac{1}{n+1}$

Монета с заданным распределением

С вероятностью $p \in (0;1)$ выпадает орёл, с вероятностью $1-p$ выпадает решка

$$r \sim U(0;1)$$

$$P(r < p) = \int\limits_0^p f_{U(0;1)}(x)dx = \int\limits_0^p dx = p$$

JumpHash: наивная реализация

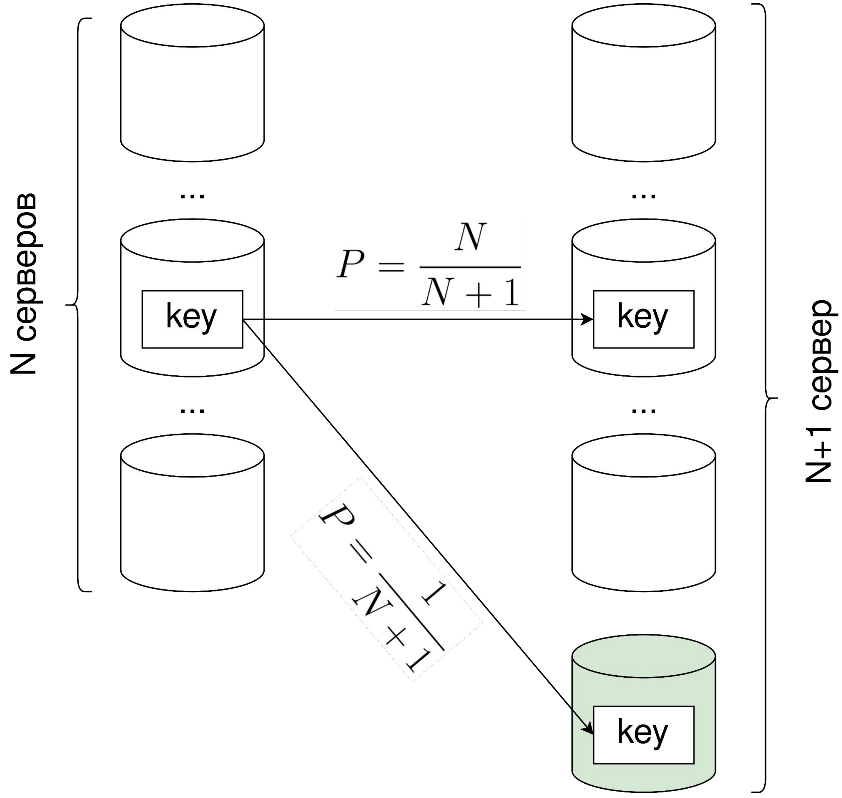
- N раз эмулируем процесс добавления нового сервера
- Подкидываем монетку и смотрим, нужно ли перемещать ключ

```
fun jump_hash(key: Key, n: Int) -> Int:
    random.set_seed(hash(key))
    result = 0
    for i = 1; i < n; i += 1:
        if random.uniform(0, 1) < 1/(i+1):
            result = i
    return result
```

- Используем псевдослучайные числа, seed инициализируем хешом ключа
  - Всегда получаем один и тот же результат для ключа key

JumpHash: оптимизация

- `ch(key, n) = ch(key, n + 1)` в большинстве случаев
- Заметим, что в большинстве случаев
- Прыжок происходят редко
- Прыжок происходит когда `ch(key, n + 1) = n + 1`



JumpHash: поиск точки следующего прыжка

- Пусть b - точка последнего прыжка
  - `ch(key, b) ≠ ch(key, b + 1)`
  - `ch(key, b + 1) = b`
- Найдём точку следующего прыжка
  - Такое максимальное `j`, что
    `ch(key, j) = ch(key, b + 1)`
- `j ≥ i (i ≥ b + 1)`, если в точке `i` не произошло прыжка
  - Если `ch(key, i) = ch(key, b + 1)`
- `P(j ≥ i) = P(ch(key, i) = ch(key, b + 1))`
- Посчитаем $P\left(ch(key, n) = ch(key, m)\right)|_{n \geq m}$

JumpHash: доказательство леммы

Докажем, что $P\left(ch(key,n) = ch(key,m)\right)\big|_{n \geq m} = \frac{m}{n}$

- $n = m : P\left(ch(key,n) = ch(key,m)\right) = 1$
- $n > m$: не должно произойти прыжков на

1. $m+1$-ом шаге (вероятность этого $1 - \frac{1}{m+1} = \frac{m}{m+1}$)
2. $m+2$-ом шаге (вероятность этого $1 - \frac{1}{m+2} = \frac{m+1}{m+2}$)
3. $\ldots$
4. n-ом шаге (вероятность этого $1 - \frac{1}{n} = \frac{n-1}{n}$)

Вероятность того, что ни одного из этих прыжков не будет, равна $\frac{m}{\cancel{m+1}} \cdot \frac{\cancel{m+1}}{\cancel{m+2}} \cdot \ldots \cdot \frac{\cancel{n-1}}{n} = \frac{m}{n}$

JumpHash: финишная прямая

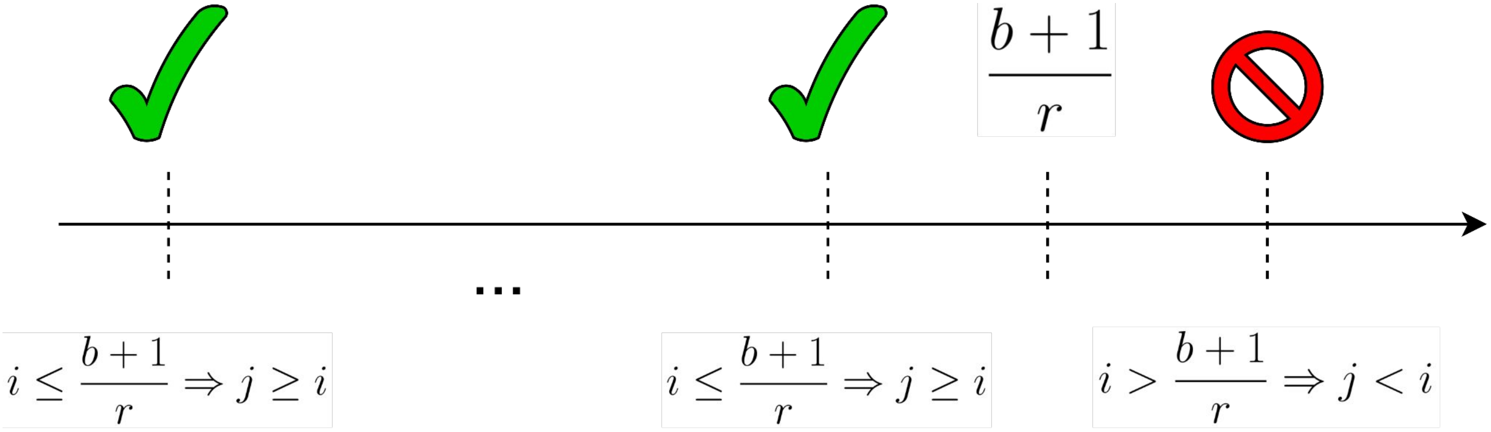
$$P\left(j \geq i\right) = P\left(ch(key, i) = ch(key, b+1)\right) = \frac{b+1}{i}$$

Сгенерируем случайное число $r \sim U(0;1)$

$j \geq i$, т. и т.т. $r \leq \frac{b+1}{i} \Rightarrow j \geq i$, т. и т.т. $i \leq \frac{b+1}{r}$

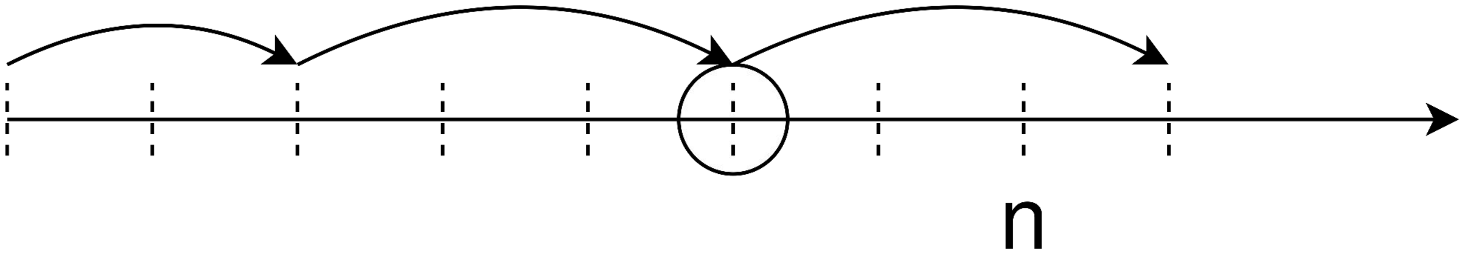
i - нижняя граница на j, поэтому $j = max\{i : i \leq \frac{b+1}{r}\}$

$j = \lfloor \frac{b+1}{r} \rfloor$



JumpHash: код алгоритма

```
fun jump_hash(key: Key, n: Int) -> Int:
    random.set_seed(hash(key))
    b = -1 # last jump point
    j = 0 # next jump point
    while j < n:
        b = j
        r = random.uniform(0, 1)
        j = ⌊(b+1)/r⌋
    return b
```



JumpHash: оценка времени работы

- Совершаем только прыжки вперёд
- На каждый узел прыгаем не более одного раза

$$\xi_i = \begin{cases} 1 & \text{если мы совершили прыжок на } i\text{-ый узел} \\ 0 & \text{иначе} \end{cases}$$

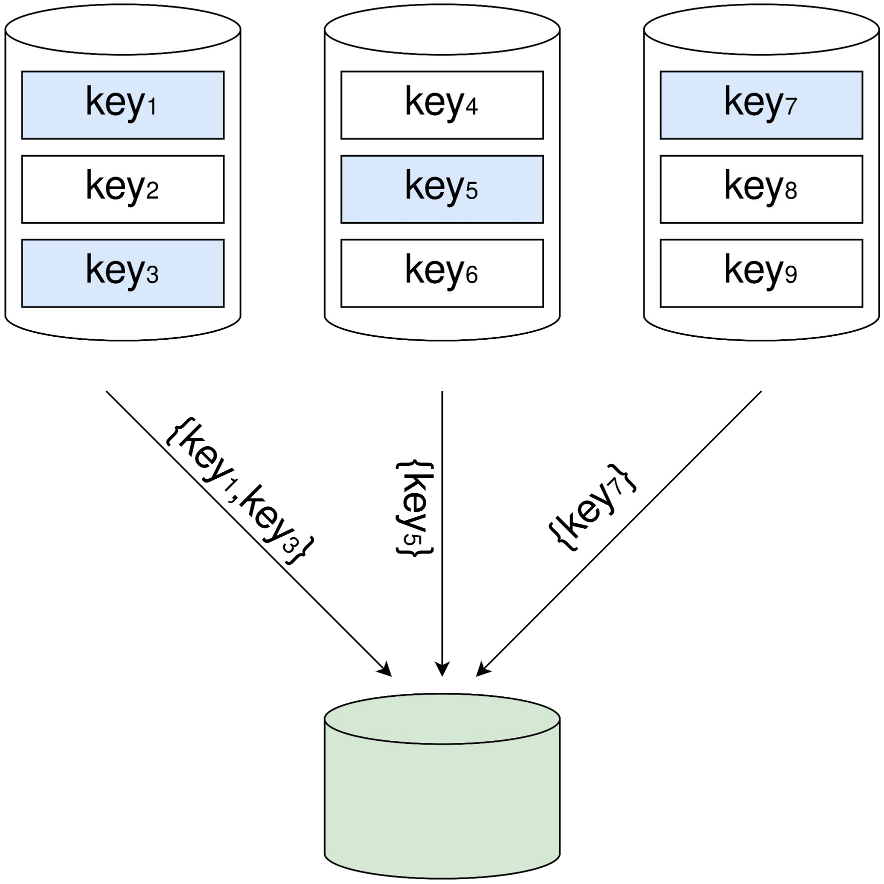
$$\mathbb{E}\,\xi_i = P\left(\text{мы совершили прыжок на } i\text{-ый узел}\right) = \frac{1}{i}$$

Оценим математическое ожидание времени работы:

$$\mathbb{E}\,T(n) = \sum_{i=1}^{n-1} \mathbb{E}\,\xi_i = \sum_{i=1}^{n-1} \frac{1}{i} = O(\log n)$$

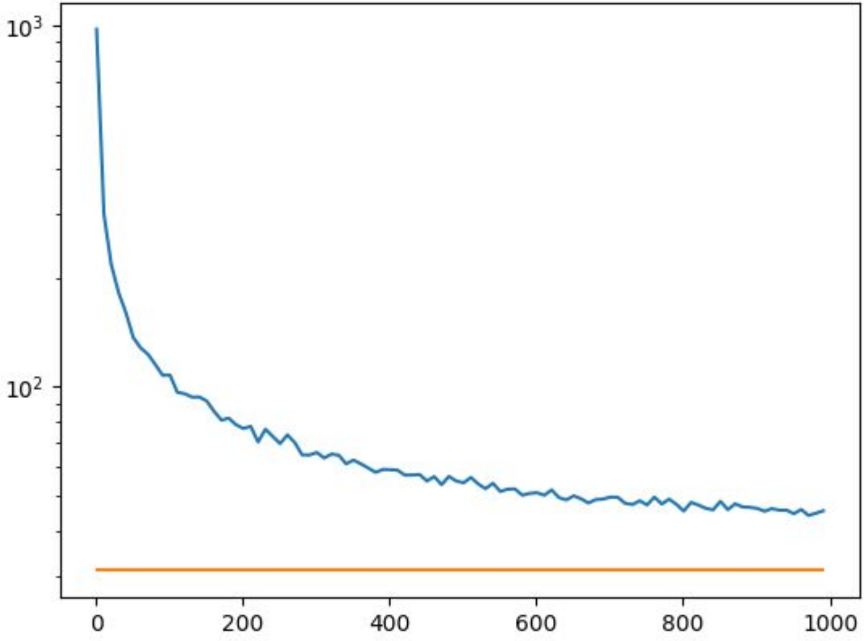
JumpHash: добавление серверов

- Сервер проходится по всем своим ключам
- Вычисляет `ch(key, n + 1)`
- Перемещает, если `ch(key, n + 1) = n + 1`



JumpHash: заключение

- Использует O(1) памяти, не использует память на хранение списка vnode
- Очень хорошо распределяет нагрузку
  - Распределение более равномерное, чем при использовании 1000 vnode на один узел в консистентном хешировании



Что почитать

- *Fagin R. et al.* Extendible hashing—a fast access method for dynamic files
- *Thaler D., Ravishankar C. V.* A name-based mapping scheme for rendezvous
- *Karger D. et al.* Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web
- *Appleton B., O'Reilly M.* Multi-probe consistent hashing
- *Lamping J., Veach E.* A fast, minimal memory, consistent hash algorithm

Thanks for your attention