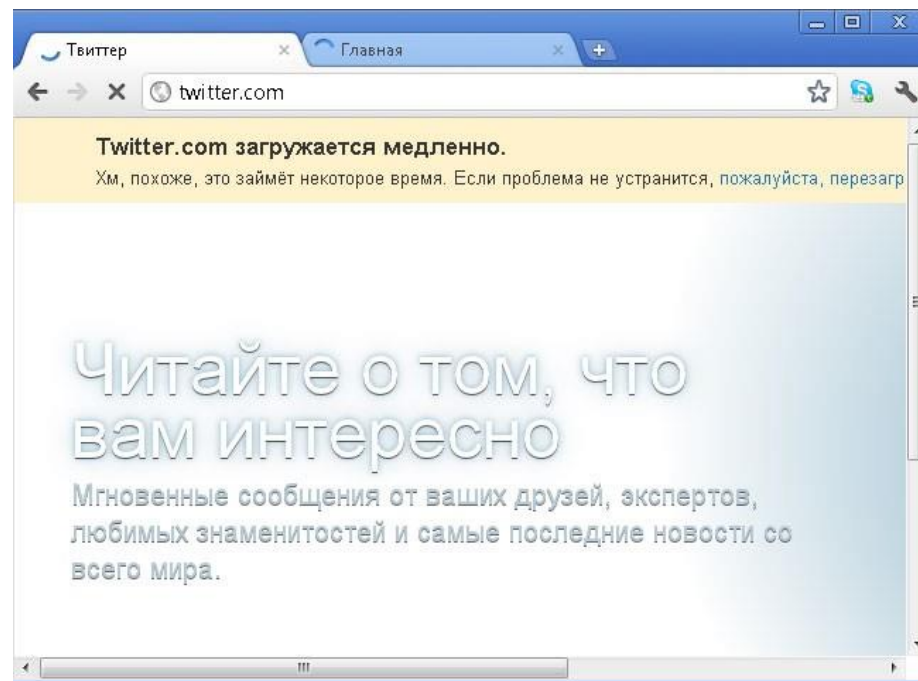




# программная инженерия

производительность

# Производительность



Производительность в значительной степени представляет собой функцию, зависящую от **частоты межкомпонентных связей** и их природы, в дополнение к тактико-техническим данным самих компонентов, и следовательно, может быть прогнозирована при изучении архитектуры системы.

# Единицы измерения производительности

# 1 Время отклика (response time)

Промежуток времени, который требуется системе, чтобы обработать запрос извне.

Основные факторы влияющие на время отклика

- Потребление ресурсов
- Время блокировки (Blocked time)
- Конкуренция за ресурсы (Contention for resources)
- Доступность ресурсов (Availability of resources)
- Зависимости от других вычислений



## 2 Быстрота реагирования (responsiveness)

- Скорость подтверждения запроса (Не путать с временем отклика— скоростью обработки !)
- Эта характеристика во многих случаях весьма важна, поскольку интерактивная система, пусть даже обладающая нормальным временем отклика, но не отличающаяся высокой быстротой реагирования, всегда вызывает справедливые нарекания пользователей
- Основные факторы влияющие на быстроту реагирования:
  - Если, прежде чем принять очередной запрос, система должна полностью завершить обработку текущего, параметры времени отклика и быстроты реагирования, по сути, совпадают
  - Если же система способна подтвердить получение запроса раньше, ее быстрота реагирования выше
  - Например, применение динамического индикатора состояния процесса копирования повышает быстроту реагирования экранного интерфейса, хотя никак не сказывается на значении времени отклика.

# Основные факторы влияющие на быстроту реагирования

- Если, прежде чем принять очередной запрос, система должна полностью завершить обработку текущего, параметры времени отклика и быстроты реагирования, по сути, совпадают
- Если же система способна подтвердить получение запроса раньше, ее быстрота реагирования выше
- Например, применение динамического индикатора состояния процесса копирования повышает быстроту реагирования экранного интерфейса, хотя никак не сказывается на значении времени отклика.

### 3 Время задержки (latency)

- Минимальный интервал времени до получения какого-либо отклика (даже если от системы более ничего не требуется)
- В телекоммуникации - время, требуемое пакету для перемещения от источника к приёмнику, сообщению - от одной точки сети к другой.
- Параметр приобретает особую важность в распределенных системах
- Снизить время задержки разработчику прикладной программы не под силу
- Фактор задержки — главная причина, побуждающая минимизировать количество удаленных вызовов

# Таблица типовых задержек

- L1 cache reference **0.5 ns**
- Branch mispredict **5 ns**
- L2 cache reference **7 ns 14x L1 cache**
- Mutex lock/unlock **25 ns**
- Main memory reference **100 ns 20x L2 cache, 200x L1 cache**
- Compress 1K bytes with Zippy **3,000 ns 3 us**
- **Send 1K bytes over 1 Gbps network 10,000 ns 10 us**
- Read 4K randomly from SSD\* **150,000 ns 150 us ~1GB/sec SSD**
- Read 1 MB sequentially from memory **250,000 ns 250 us**
- Round trip within same datacenter **500,000 ns 500 us**
- Read 1 MB sequentially from SSD\* **1,000,000 ns 1,000 us 1 ms ~1GB/sec SSD, 4X memory**
- Disk seek **10,000,000 ns 10,000 us 10 ms 20x datacenter roundtrip**
- Read 1 MB sequentially from disk **20,000,000 ns 20,000 us 20 ms 80x memory, 20X SSD**
- Send packet CA->Netherlands->CA **150,000,000 ns 150,000 us 150 ms**



## 4 Пропускная способность

Количество данных (операций),  
передаваемых выполняемых в единицу  
времени;

Пример:  
120 сообщений в час (не гарантирует 2  
сообщения в минуту).

В корпоративных приложениях  
обычной мерой производительности  
служит число транзакций в секунду  
(transactions per second — tps)

- Транзакции различаются по степени сложности
- Для конкретной системы необходимо рассматривать смесь "типовых" транзакций

## 5 Рабочая нагрузка (Work load)

Значение, определяющее степень "давления" на систему:

- Общее количество пользователей
- Количество одновременно активных пользователей
- Объемы данных
- Количество транзакций

Примеры:

- 100 пользователей работают одновременно в системе.
- 10 пользователей размещают заказы одновременно.

Параметр загрузки обычно служит контекстом для представления других функциональных характеристик. Так, нередко можно слышать выражения наподобие следующего:

«время отклика на запрос составляет 0,5 секунды для 10 пользователей и 2 секунды для 20 пользователей»

## 6 Чувствительность к загрузке (load sensitivity)

Выражение, задающее зависимость времени отклика от загрузки

Пример:

- Система А обладает временем отклика, равным 0,5 секунды для 10-20 пользователей
- Система В обладает временем отклика в 0,2 секунды для 10 пользователей и 2 секунды для 20 пользователей
- Это дает основание утверждать, что система А обладает меньшей чувствительностью к загрузке

# 7 Эффективность (efficiency)

Удельная производительность в пересчете на одну единицу ресурса

Например, система с двумя процессорами, способная выполнить 30 tps, более эффективна по сравнению с системой, оснащенной четырьмя аналогичными процессорами и обладающей продуктивностью в 40 tps



## 8 Мощность (Capacity)

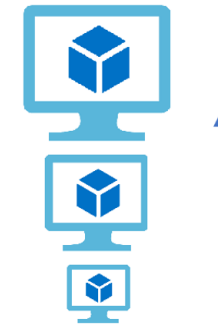
- Наибольшее значение пропускной способности или загрузки.
- Это может быть как абсолютный максимум, так и некоторое число, при котором величина производительности все еще превосходит заданный приемлемый порог.

## 9 Способность к масштабированию (scalability)

- Свойство, характеризующее поведение системы при добавлении ресурсов (обычно аппаратных)
- Масштабируемой принято считать систему, производительность которой возрастает пропорционально объему приобщенных ресурсов (скажем, вдвое при удвоении количества серверов)

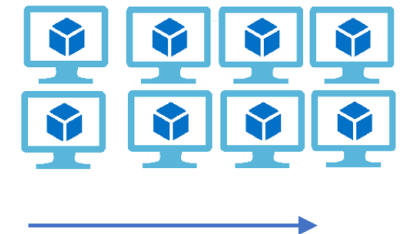
Vertical Scaling

( Increase size of instance (RAM , CPU etc.) )

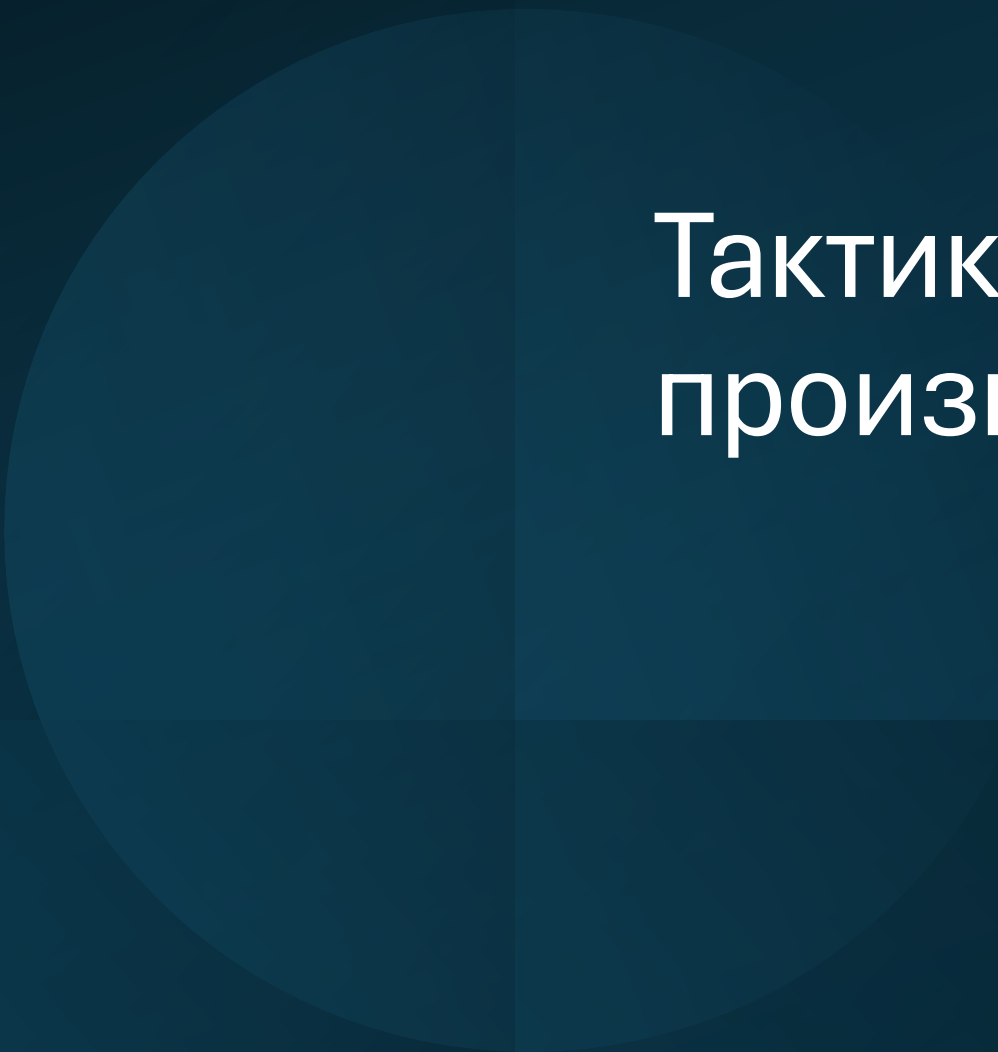


Horizontal Scaling

( Add more instances )



[www.abhijitkakade.com](http://www.abhijitkakade.com)

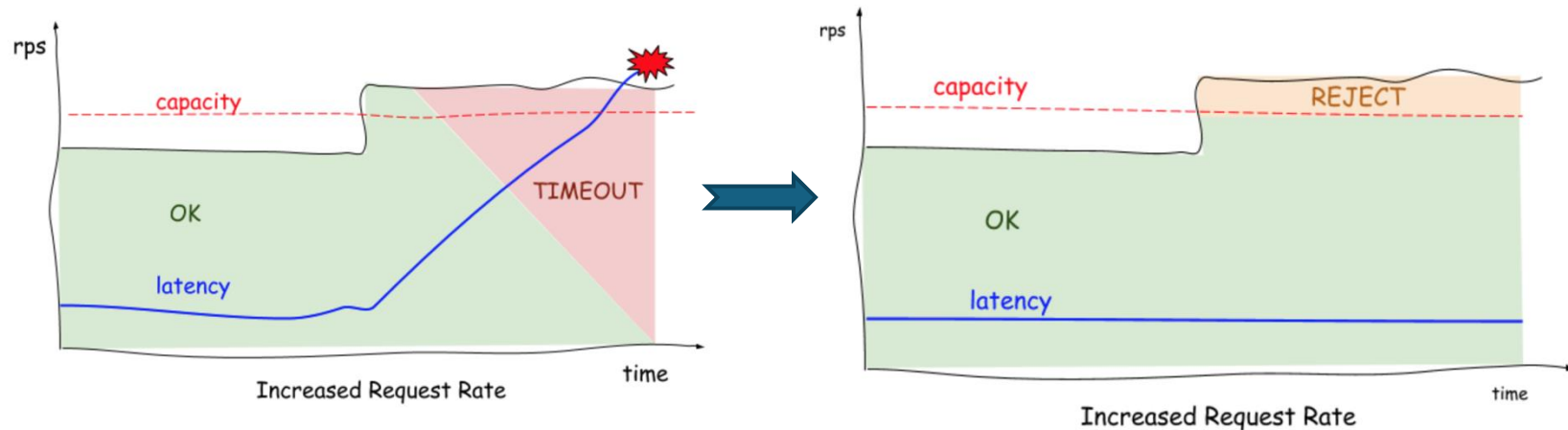


# Тактики управления производительностью

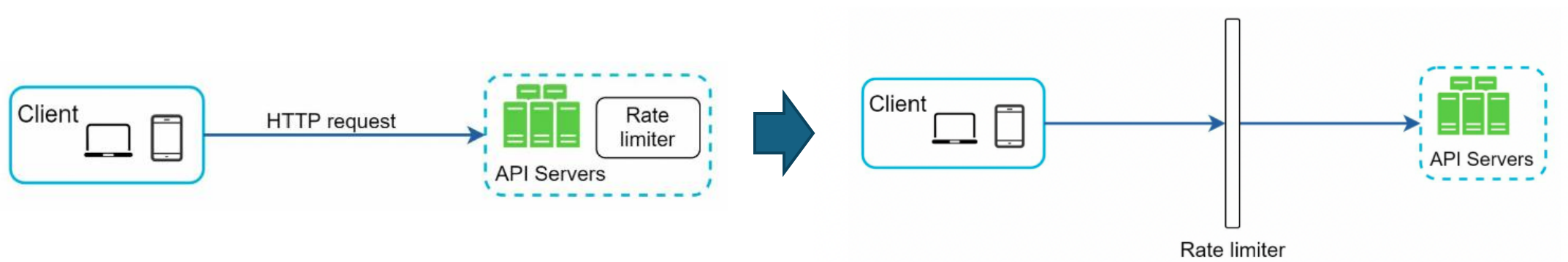
# 1 Верхняя граница загрузки

Верхняя граница загрузки ресурса (лишние запросы не выполняются, но остальные выполняются с требуемой скоростью)

- Ограничение время выполнения. Иногда имеет смысл установить ограничение на время обработки запроса.
- Ограничение размера очереди. Заранее определяются максимальные размеры очереди и число ресурсов требуемых для обработки очереди.

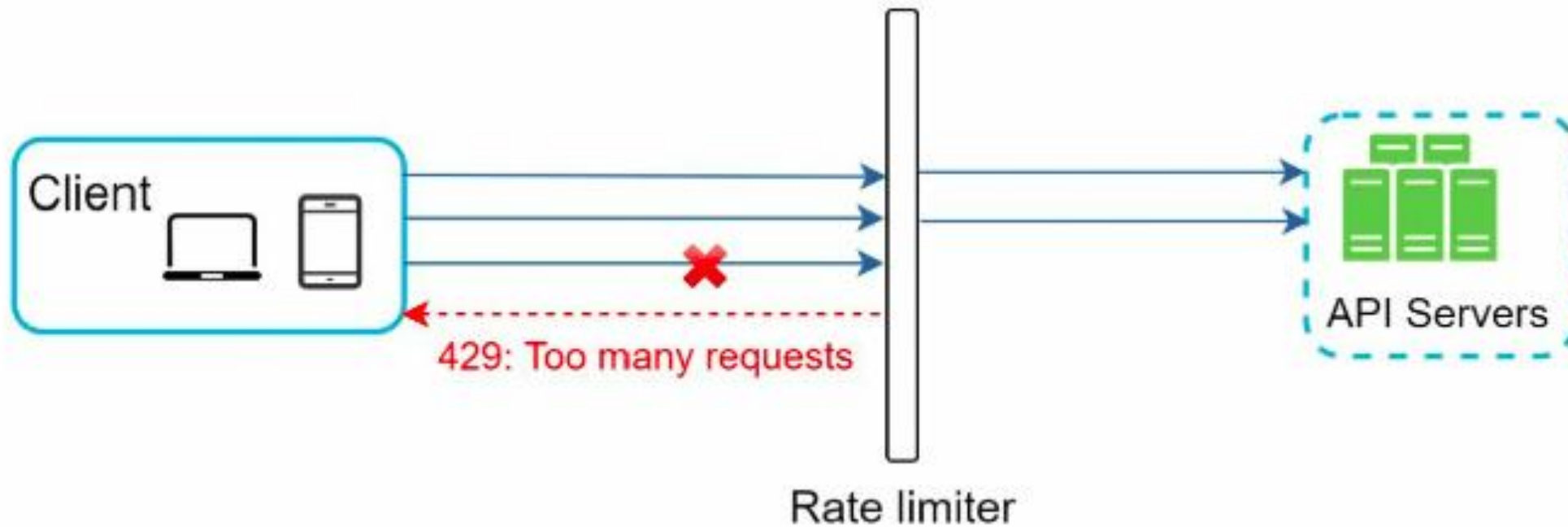






# Rate Limiter

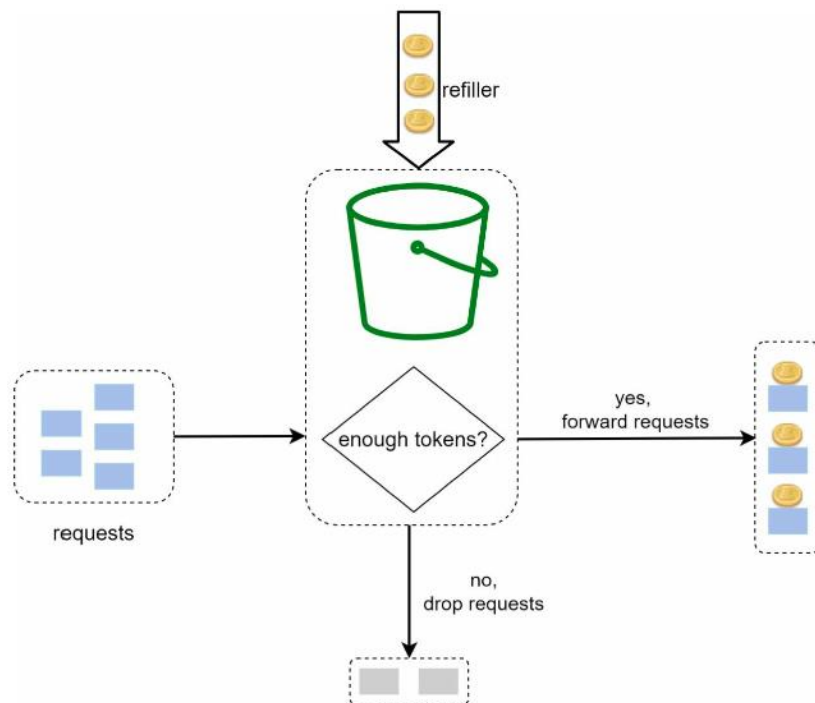
# Обработка превышения количества запросов



# Алгоритмы rate limiter

1. Token bucket
2. Leaking bucket
3. Fixed window counter
4. Sliding window log
5. Sliding window counter

# Token bucket



У корзины есть максимальное количество токенов которое может в нее вместиться

Токены добавляются с predetermined интенсивностью

Если корзина полна – токены не добавляются

Каждый запрос – забирает один токен

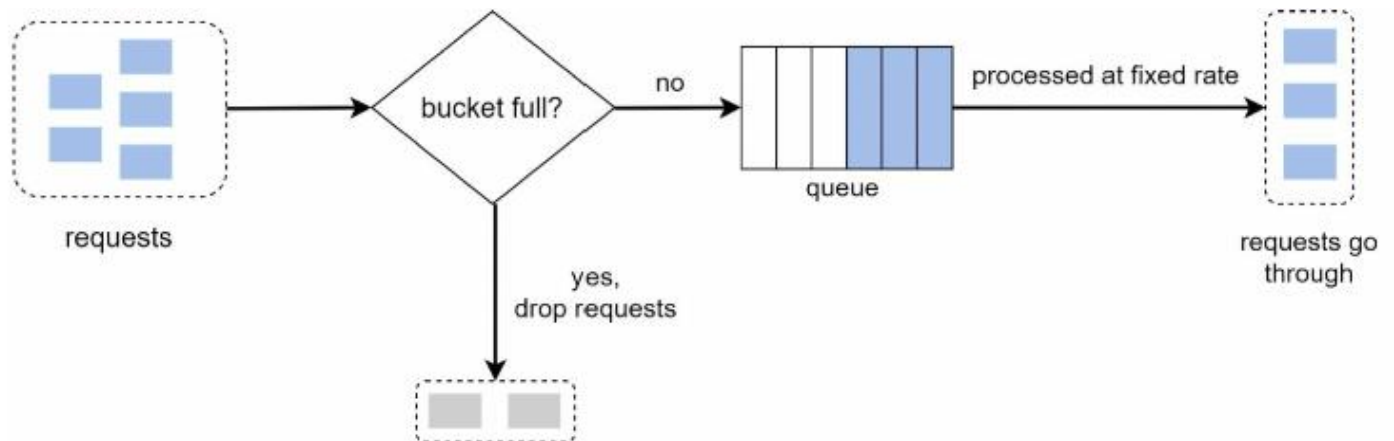
Если токенов нет – происходит отказ в обработке запроса

**Простой, эффективные по памяти, дает возможность обрабатывать локальные пики нагрузки**

# Leaking bucket

Аналогичен Token bucket за несколькими исключениями

- Запросы обрабатываются с заданной скоростью по принципу FIFO
- Размер очереди – размеру корзины
- Вместо токенов – «свободные места в очереди»



Локальный скачек трафика может забить очередь «старыми запросами», а новые не будет обрабатываться

# Fixed window counter

Делим время на отрезки фиксированной длины (например по секунде)

Для каждого отрезка определяется counter считающий количество запросов

Если counter превысил допустимый максимум – то запрос отклоняется

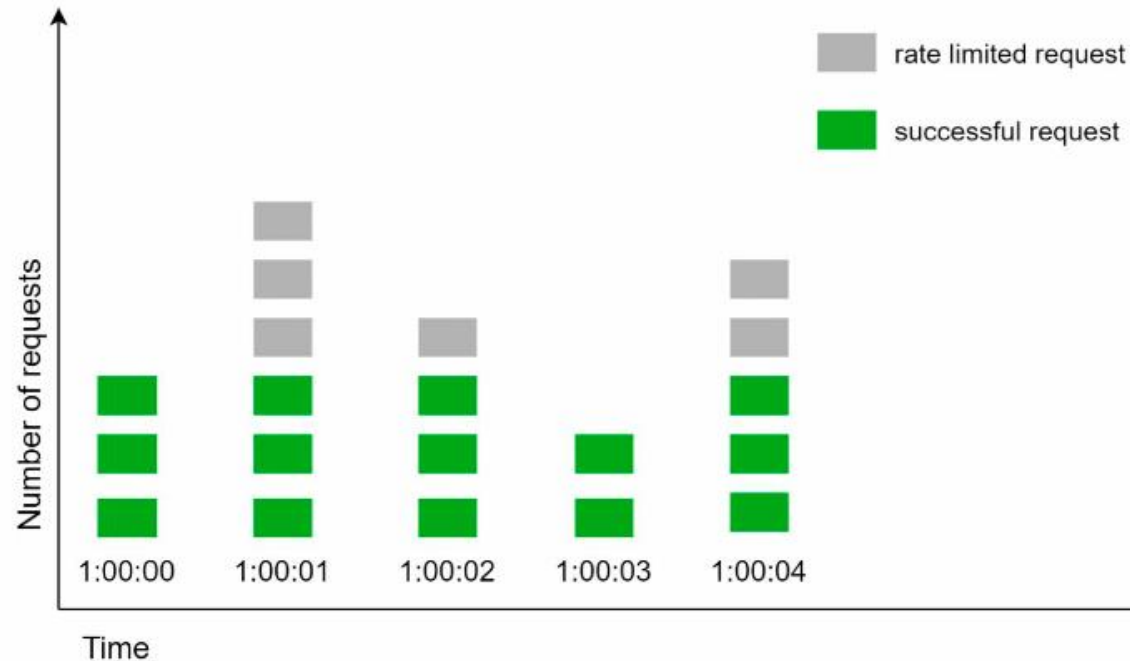
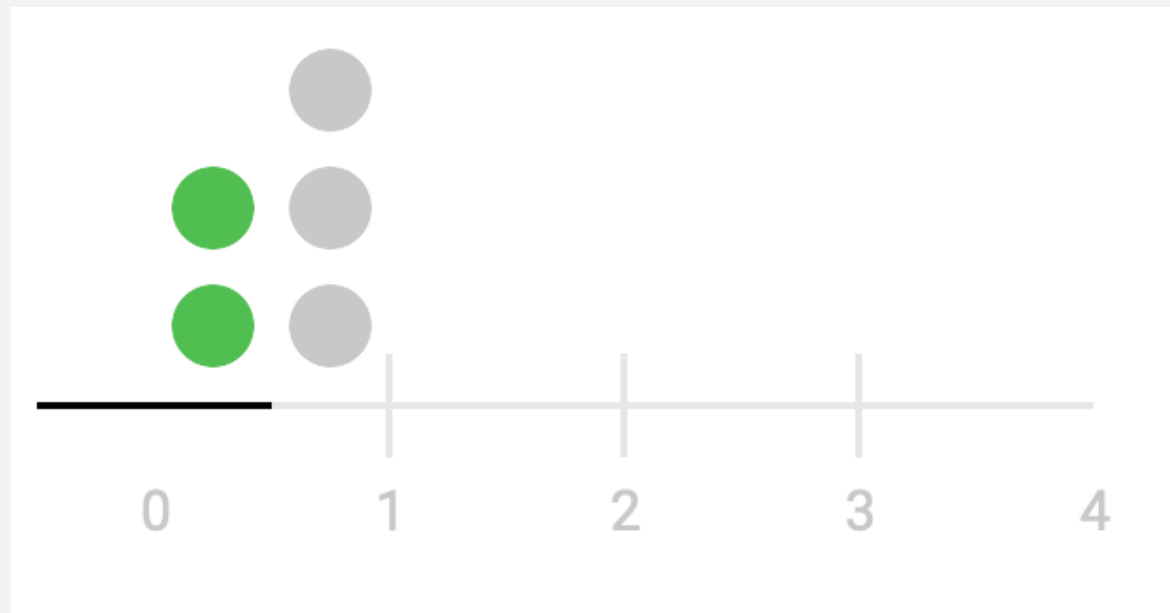


Figure 4-8

На границах окон может быть локальный скачек

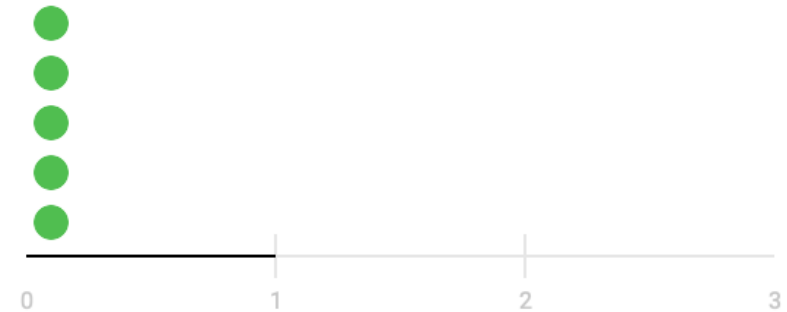
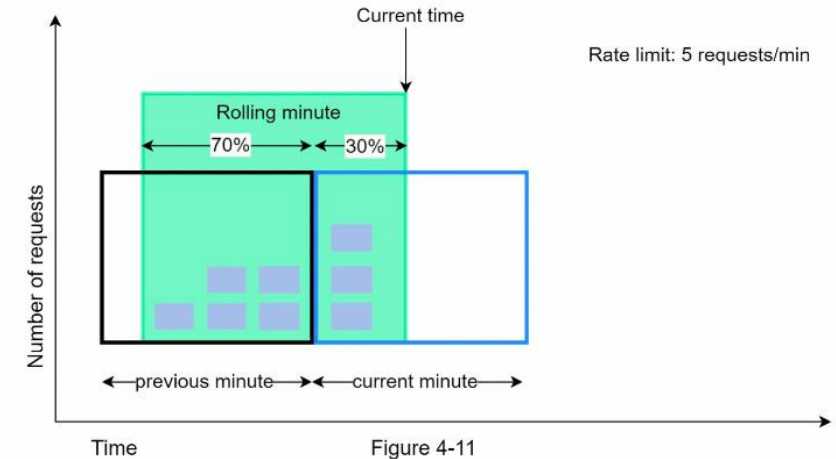
# Sliding window log



- 
- Улучшение Fixed window algorithm
  - Запоминаем timestamp запросов и сохраняем их в кеше
  - Когда приходит новый запрос – удаляем все «устаревшие запросы». Устаревшие – те которые не принадлежат текущему окну
  - Новые timestamp запросов – помещаем в кеш
  - Если размер лога запросов меньше чем определенный размер – то запрос проходит (считаем только прошедшие запросы)

# Sliding window counter

- Количество запросов считается как «количество запросов в текущем окне» + «количество запросов в пересекающей части скользящего окна»\*(процент размера скользящего окна)
- $3 + 5 * 0.7 = 6.5$

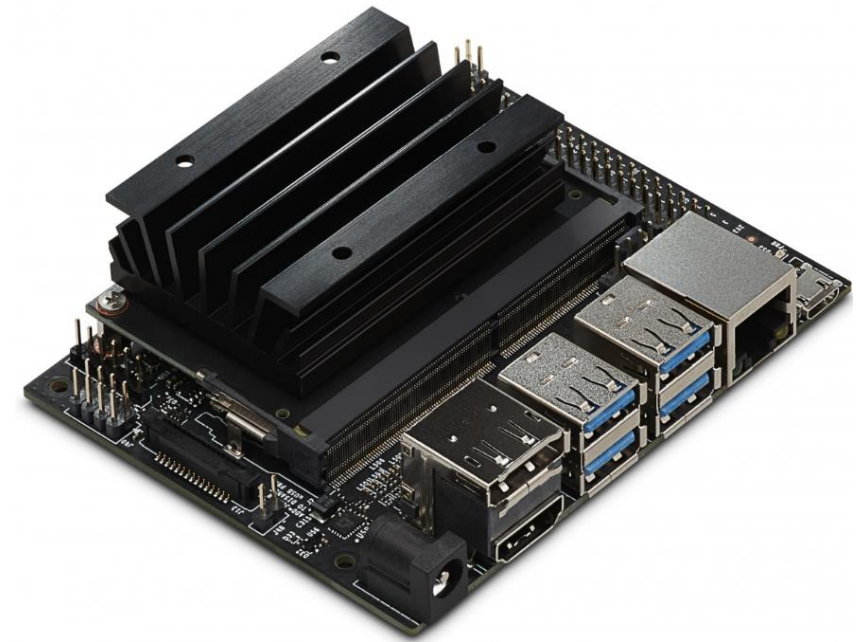




## 2 Более быстрое оборудование

### **Увеличение доступности ресурса.**

Более быстрые процессоры, дополнительные процессоры, больше памяти, более быстрая сеть ....



# 3 Управление потреблением ресурсов

Необходимо уменьшить число запросов к медленным ресурсом в ходе работы программы.

Способы:

1. **Увеличение эффективности** вычислений (основной подход). Например: переход от сортировки пузырьком к сортировке Хоара
2. **Масштабирование**  
Увеличиваем количество ресурсов, тем самым уменьшаем нагрузку
3. **Кэширование**  
Заменяем медленные ресурсы – быстрыми.

# 1 Увеличение эффективности

Различные  
алгоритмы  
сортировок  
(обращения к  
памяти)

	 Insertion	 Selection	 Bubble	 Shell	 Merge	 Heap	 Quick	 Quick3
Random								
Nearly Sorted								
Reversed								
Few Unique								

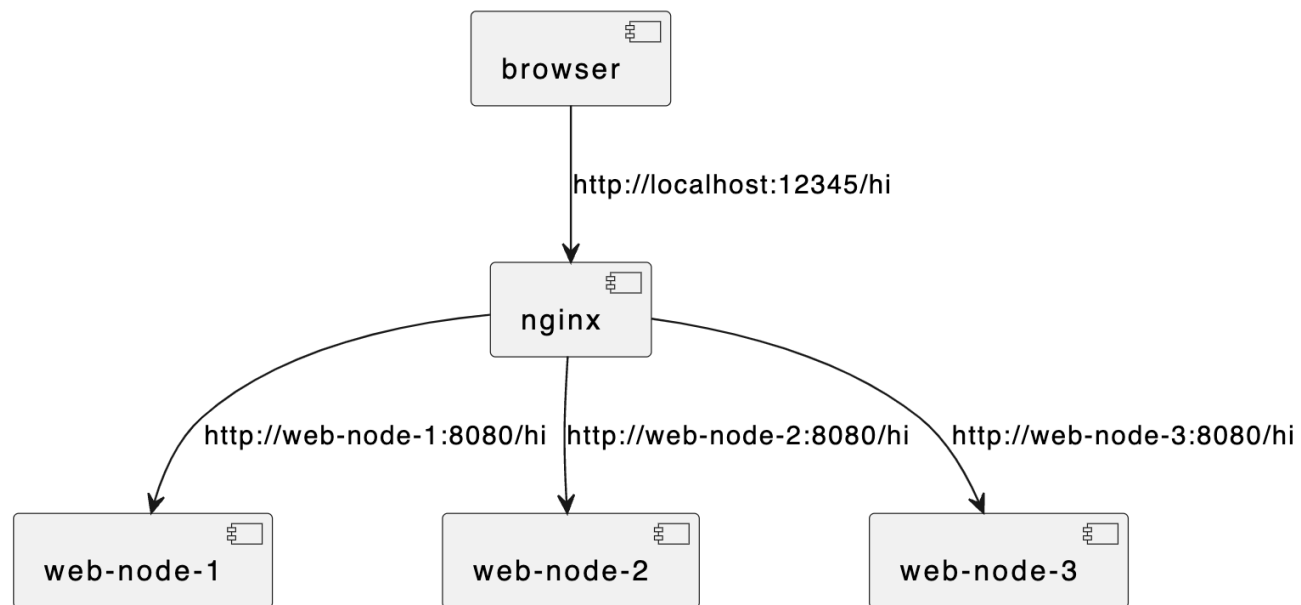
# 2 Масштабирование

/etc/nginx/conf.d/server.conf

```
## Конфигурация nginx

upstream app {
    server web-node-1:8080;
    server web-node-2:8080;
    server web-node-3:8080;
}

server {
    listen 80;
    location / {
        proxy_pass http://app;
    }
}
```

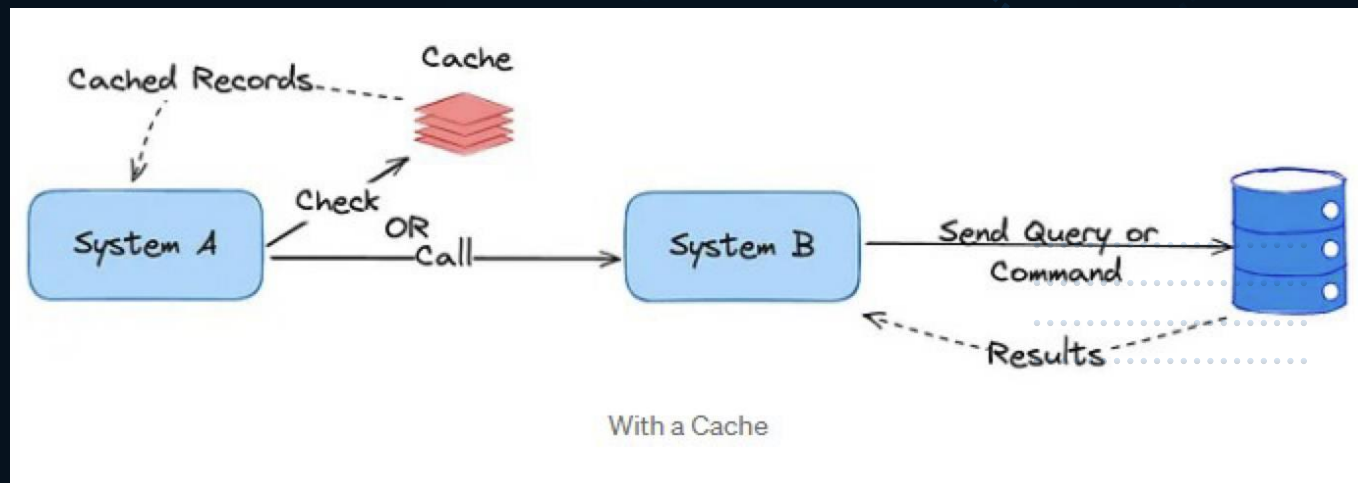
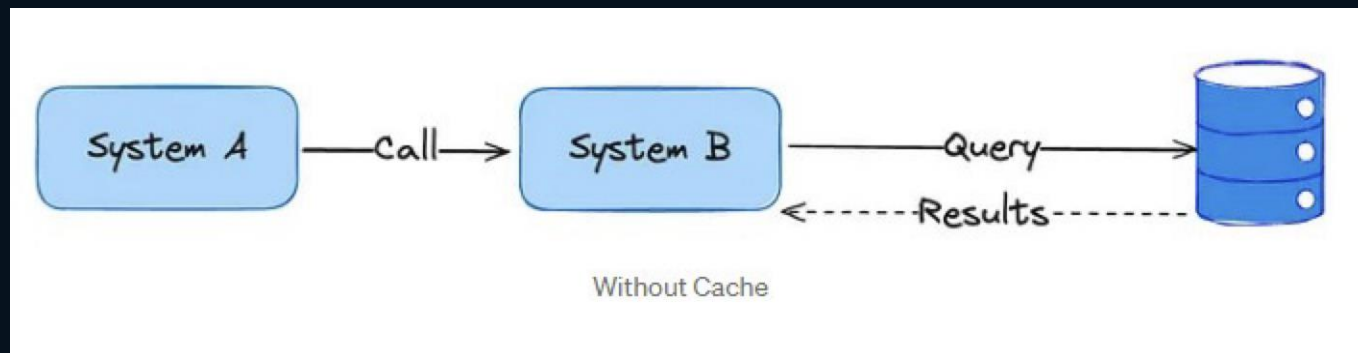


## 3 Кэш

Кэш или кеш(англ. cache, от фр. cacher—«прятать»; произносится [kæʃ] —«кэш») —промежуточный буфер с быстрым доступом к нему, содержащий информацию, которая может быть запрошена с наибольшей вероятностью.

Доступ к данным в кэше осуществляется быстрее, чем выборка исходных данных из более медленной памяти или удалённого источника, однако её объём существенно ограничен по сравнению с хранилищем исходных данных.

# кеширование



# Принцип локальности

В локальные моменты времени используется лишь небольшое подмножество данных.

- Для чатов - последние сообщения
- Для новостей - последние новости
- Для сервиса такси - активные заказы.

# Кэширование в распределенных приложениях

- ✓ В распределенных приложениях есть два основных типа стратегий кэширования данных, которые вы можете использовать
  1. Частный кэш
  2. Общий кэш
- ✓ Вы можете использовать обе стратегии в одном приложении. Некоторые данные могут храниться в частном кеше, тогда как другие данные могут храниться в общем кеше.

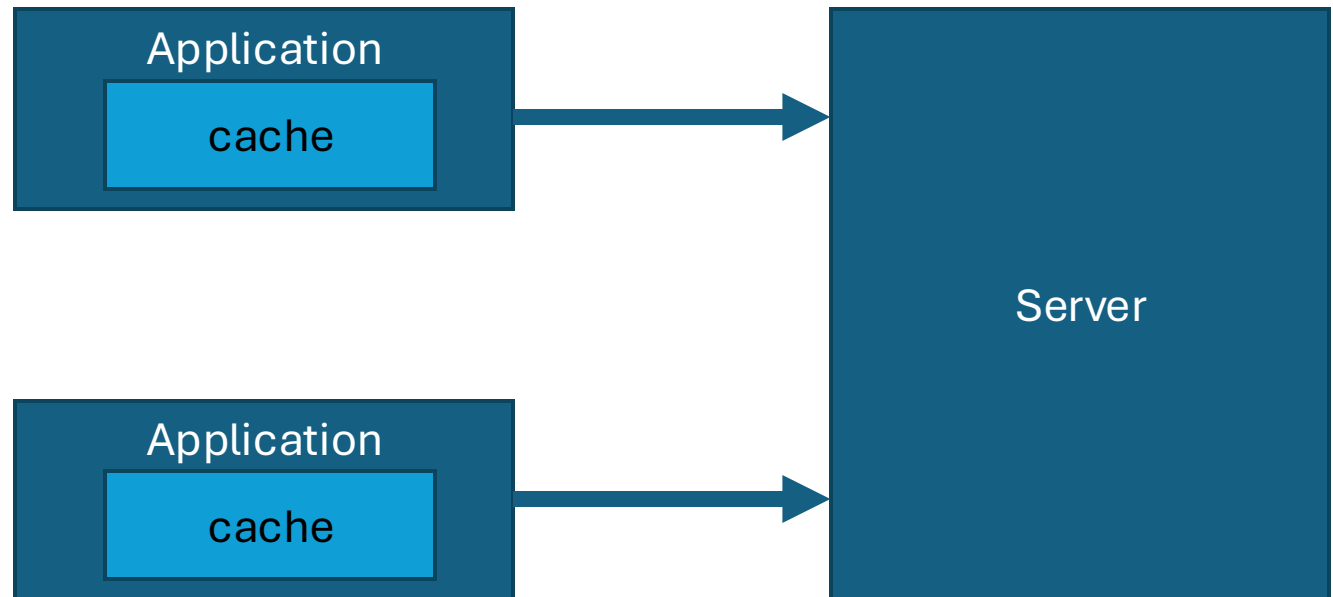


# 1 Частный кэш

- Частный кэш хранится **на компьютере, на котором запущено приложение**, которое его использует.
- Если на одном компьютере запущено несколько экземпляров приложения, то **каждый экземпляр приложения может иметь свой собственный кэш**.
- Одним из способов хранения данных в частном кэше является оперативная память, что делает его чрезвычайно быстрым. Если необходимо кэшировать больше данных, чем может поместиться в объеме памяти, доступной на машине, то кэшированные данные могут храниться в локальной файловой системе.
- В распределенной системе, использующей стратегию частного кэширования, каждый экземпляр приложения будет иметь свой собственный кэш. Это означает, что один и тот же запрос может давать разные результаты в зависимости от экземпляра приложения.

## 1.1 Частный кэш на клиенте

- у каждого клиента свой
- позволяет ускорить запрос к редко меняющимся данным
- кэширует ответ сервера целиком

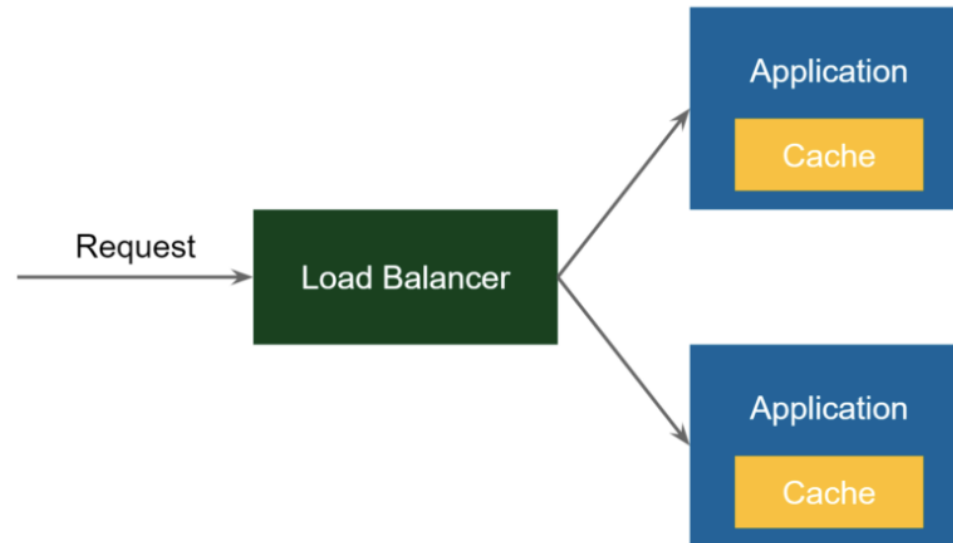


# Серверное кэширование

- Кэширование включает в себя **копирование данных, которые могут понадобиться снова**, в быстрое хранилище, чтобы к ним можно было быстрее получить доступ при последующих использованиях.
- Кэши на стороне сервера можно использовать, чтобы **избежать многократного дорогостоящего поиска данных из исходного хранилища данных** (например, реляционной базы данных).
- Кэш на стороне сервера должен располагаться как можно ближе к приложению, чтобы минимизировать задержки и сократить время отклика.
- Тип хранилища, используемого для кэша на стороне сервера, предназначен для быстрой работы, такой как база данных в памяти. Чем больше данных и пользователей приходится обрабатывать приложению, тем больше преимуществ кэширования.

## 1.2 Частный кэш на сервере

- кэш хранится на уровне сервера
- все кэши изолированы друг от друга
- приходится кэшировать одно и то же много раз на каждом из экземпляров сервера

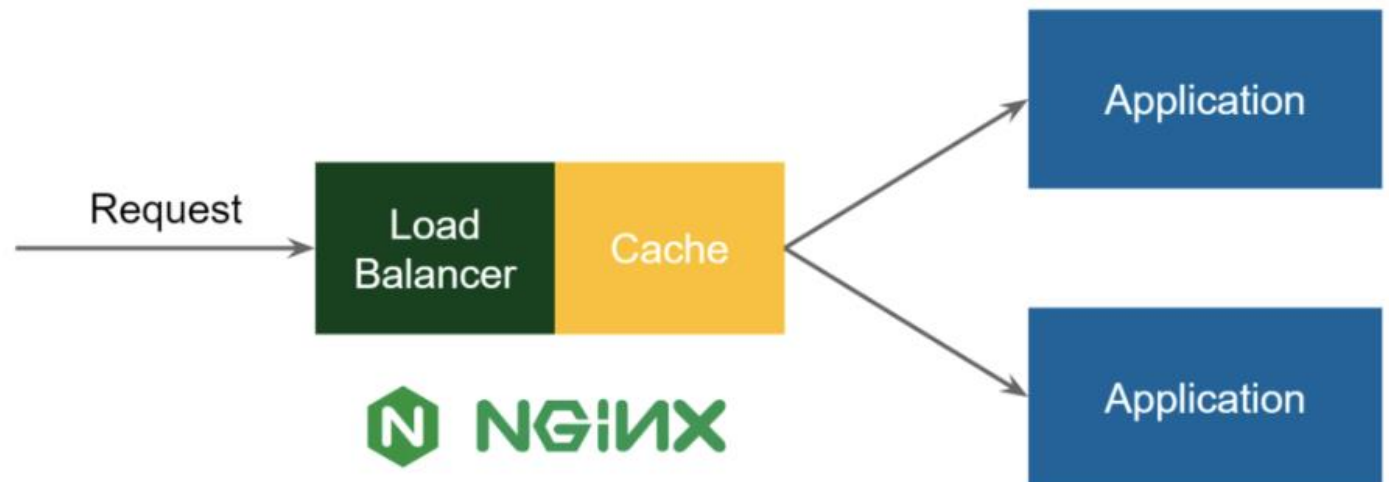


## 2. Общий кэш

- Все экземпляры приложений используют общий кэш. Это решает проблему разных экземпляров приложения, потенциально имеющих разные представления кэшированных данных.
- Общий кэш **работает медленнее, чем частный**, потому что вместо того, чтобы быть доступным на том же компьютере, что и экземпляр приложения, он находится где-то еще; при взаимодействии с кешем будут возникать задержки.
- Виды:
  1. Reverse Proxy
  2. Standalone cache
  3. Distributed cache
  4. Sidecar cache

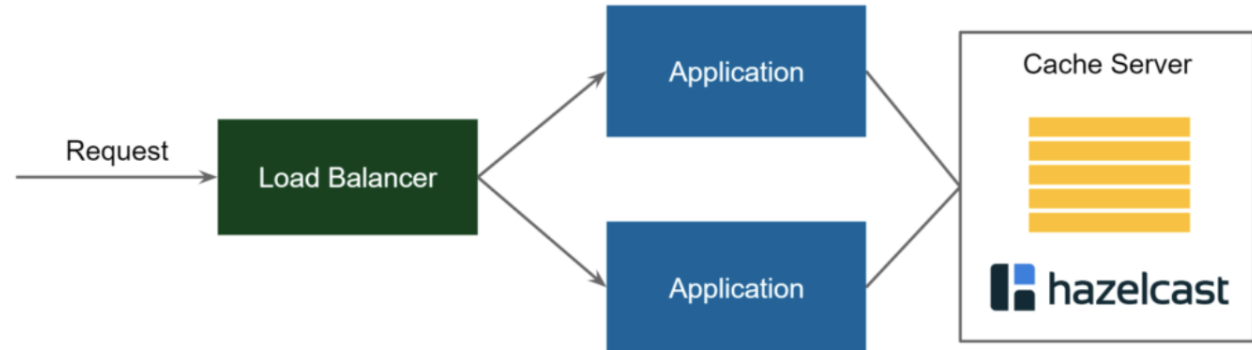
## 2.1 Reverse proxy

- Кэш хранится на балансере
- Инвалидация кэша на стороне сервера, а не приложения



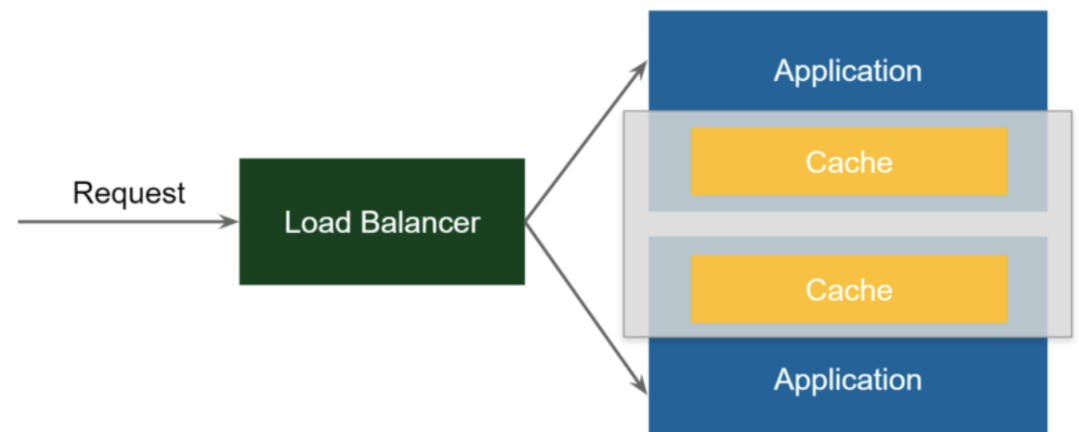
## 2.2 Standalone cache

- Кэш хранится в отдельном сервисе
- Сервис кэша можно отдельно скейлить, если надо
- Стек приложения не важен



## 2.3 Distributed cache

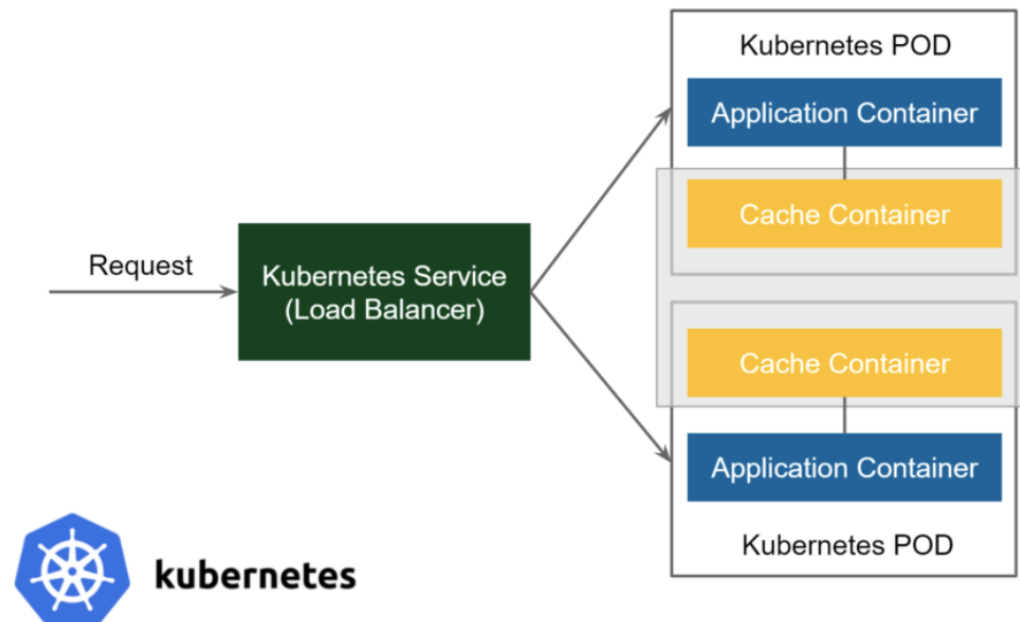
- кэш хранится на уровне приложения, но шарится на несколько приложений (реплицируется или распределяется)
- Зависит от реализации в фреймворке и языке программирования





## 2.4 Sidecar КЭШ

- кэш хранится на приложения кэша, но шарится на несколько приложений инстансов
- Может работать в режиме репликации, так и в распределенном режиме
- Ниже latency
- Не зависит от стека приложения

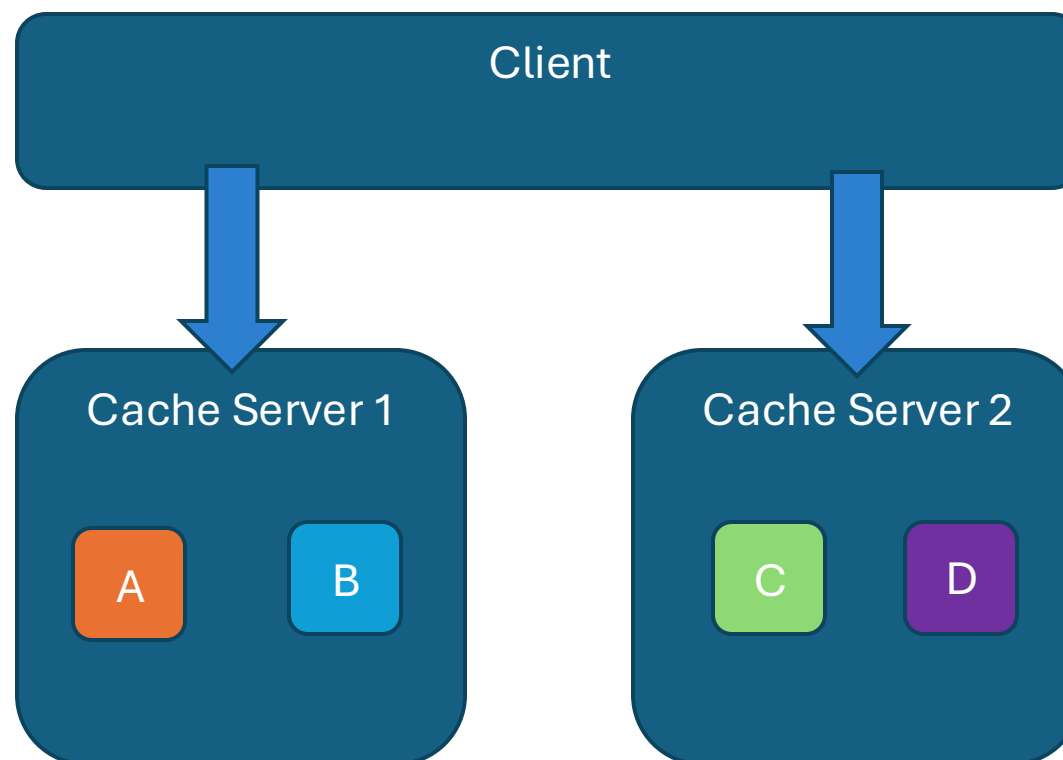


# Как организуются данные в распределенных кэшах?

Sharded vs Replicated

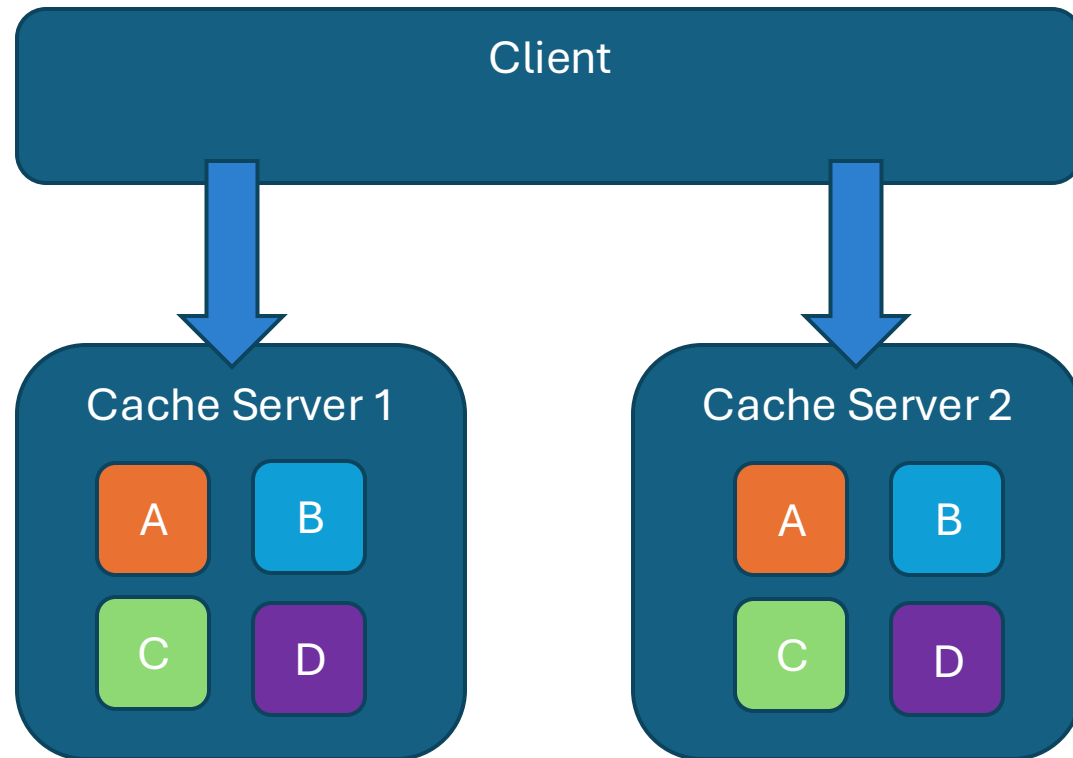
# Sharded cache

- Данные распределяются между экземплярами (sharding)
- Если ключ не в текущем шарде, идем в другой инстанс



# Replicated cache

- Рреплицируется на все экземпляры.
- Быстрее распределенного, но требует больше ресурсов на экземп



# Задачи по работе с кэшем

прогрев, обновление, устаревание, вытеснение

# Пред-загрузка кэша (прогревание)

- Пока в кэше нет данных – он не ускоряет работу.
- Это означает, что приложение предварительно заполняет кэш при запуске приложения данными, **которые либо понадобятся при запуске, либо используются достаточно часто**, чтобы данные были доступными в кэше с самого начала.
- Это может помочь повысить производительность, с самых первых запросов к серверу.

# Обновление данных кэша

- **Данные в кеше могут устареть**, если их изменить после помещения в кеш.
- **Аннулирование кэша - это процесс замены или удаления кэшированных элементов.** Мы должны обеспечить правильную обработку кэшированных данных, чтобы устаревшие данные заменялись или удалялись. Также может потребоваться удалить кэшированные элементы, если кэш заполнится.



# Устаревание данных кэша

Когда данные кэшируются, мы можем настроить **истечение срока действия** данных из кэша через определенный промежуток времени.



# Алгоритмы вытеснение из кэша

1. **Least recently used (LRU):**  
Исходим из предположения о том, что недавно использовавшиеся кэшированные элементы наиболее вероятно будут использованы снова в ближайшее время, и удаляем элементы, которые использовались давно первыми.
2. **Most recently used (MRU):**  
исходя из предположения, что недавно использовавшиеся кэшированные элементы больше не понадобятся, удаляем элементы которые использовались в последнее время первыми.
3. **First-in, first-out (FIFO).**  
Как и очередь FIFO, здесь отбрасывается элемент, который был помещен в кэш первым (самые старые данные).
4. **Last-in, first-out (LIFO).**  
Этот подход противоположен FIFO в том смысле, что он отбрасывает элемент, который был помещен в кэш в последний раз (самые новые данные).
5. **Явное удаление данных:**  
бывают случаи, когда мы хотим явно удалить данные из кэша, например, после удаления или обновления существующих данных.

# Принцип

- Вы должны держать нагрузку без кеша.
- Задача кеша-ускорить ответ, а не держать нагрузку. Проводите учения. **Убедитесь, что вы работаете без кеша.**

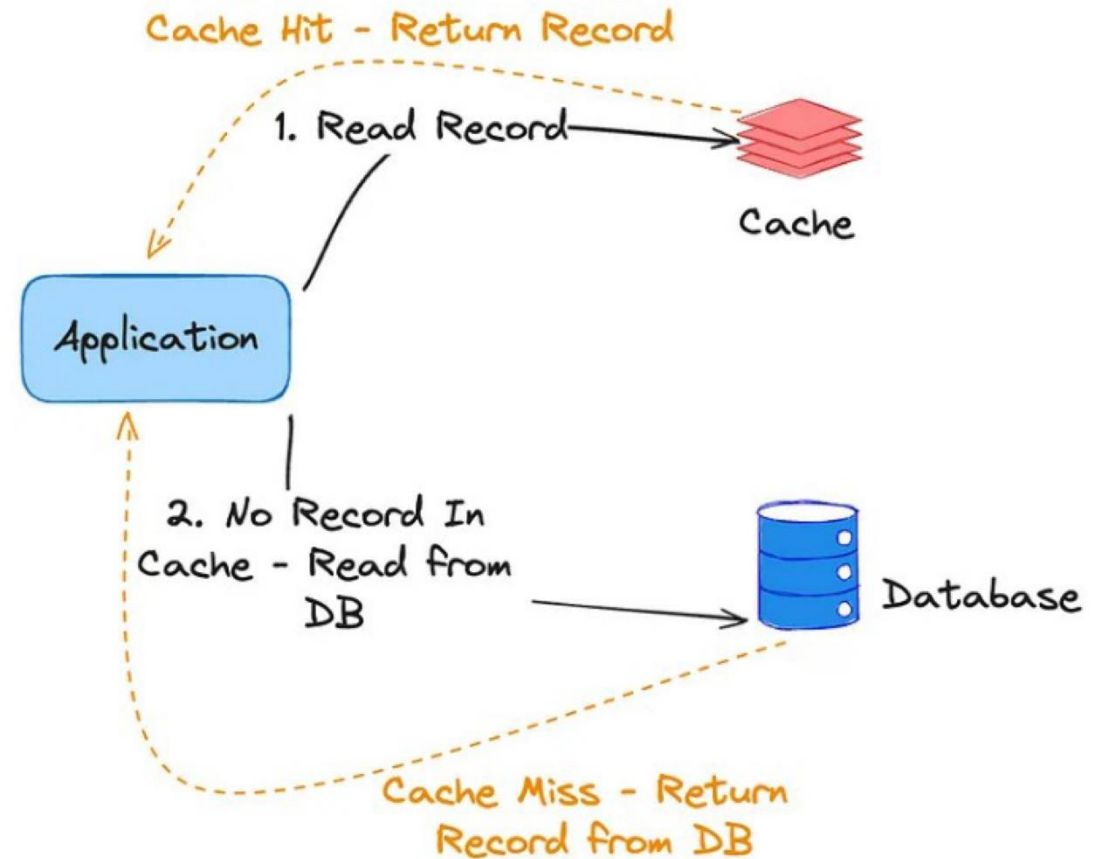
# Cache-miss

AverageTime= DbAccessTime\*  
CacheMissRate+ CacheAccessTime

Пусть:

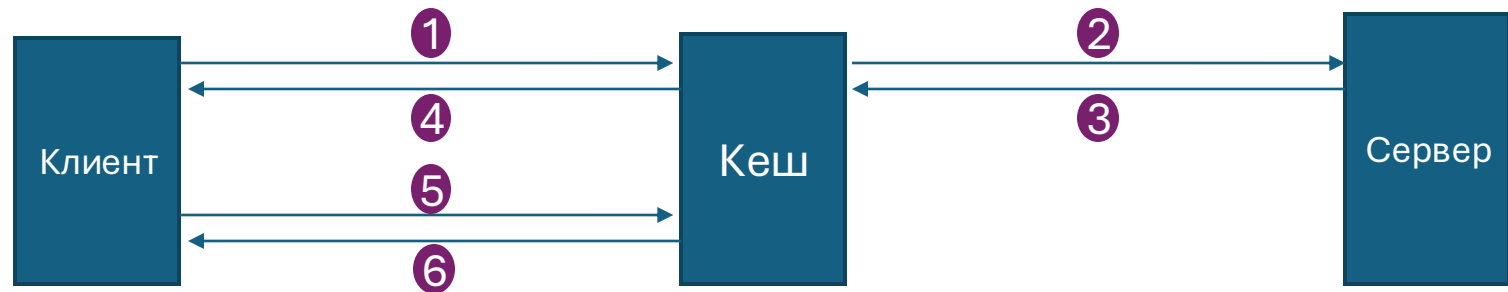
- DbAccessTime= 100ms
- CacheAccessTime= 20ms

Тогда при CacheMissRate> 0.8 –кеш вреден!



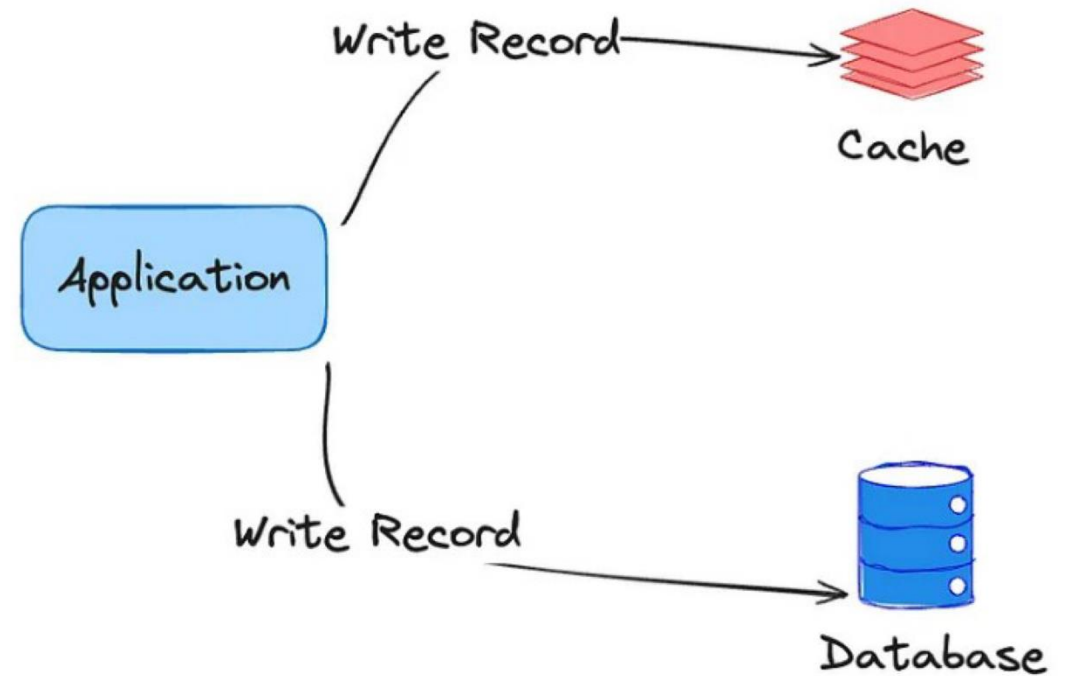
# Шаблон «СКВОЗНОЕ ЧТЕНИЕ»

- При использовании шаблона сквозного чтения кэш работает как замена системы-хранилища и содержит компонент, который может загружать данные из фактической системы-хранилища (базы данных).
- Когда приложение запрашивает данные, система кеша пытается получить их из кеша.
- Если он не существует в кэше, он извлекает данные из хранилища, сохраняет их в кэше и возвращает их.

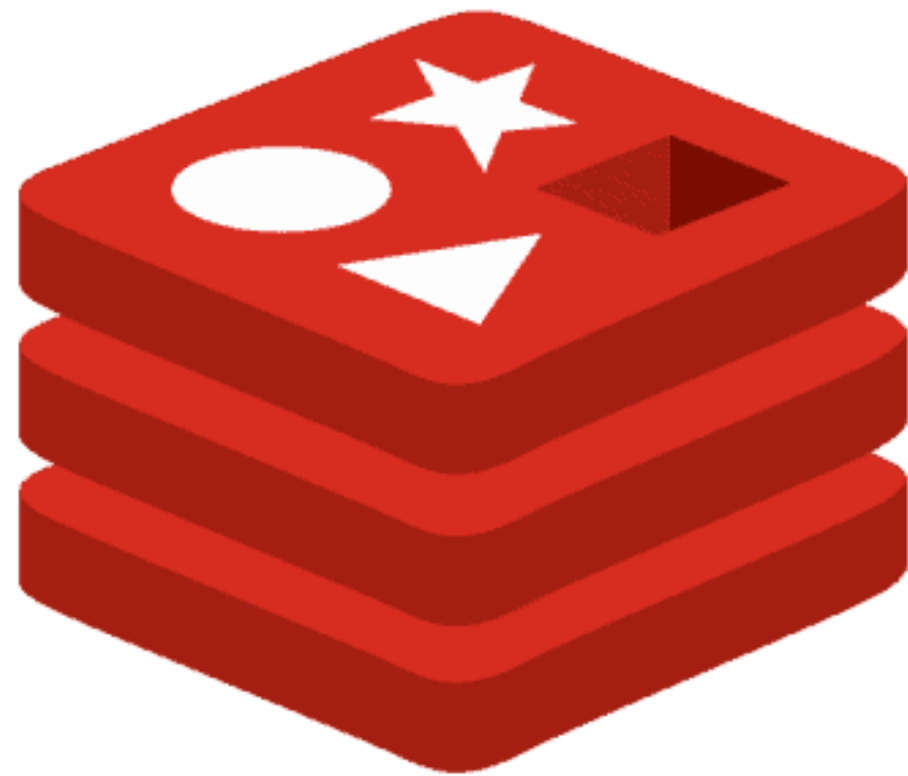


## Шаблон «сквозная-запись»

- Система кэширования, использующая шаблон сквозной записи, имеет компонент, который может записывать данные в систему хранения данных.
- Приложение работает с кэшом как с системой записи, и когда оно запрашивает систему кэширования для записи данных, оно записывает данные в систему записи (базу данных) и обновляет кэш.



Redis



Name	Type	Data storage options	Query types	Additional features
Redis	In-memory non-relational database	Strings, lists, sets, hashes, sorted sets	Commands for each data type for common access patterns, with bulk operations, and partial transaction support	Publish/Subscribe, master/slave replication, disk persistence, scripting (stored procedures)
memcached	In-memory key-value cache	Mapping of keys to values	Commands for create, read, update, delete, and a few others	Multithreaded server for additional performance
MySQL	Relational database	Databases of tables of rows, views over tables, spatial and third-party extensions	SELECT, INSERT, UPDATE, DELETE, functions, stored procedures	ACID compliant (with InnoDB), master/slave and master/master replication



# Типы данных

**Table 1.2** The five structures available in Redis

Structure type	What it contains	Structure read/write ability
STRING	Strings, integers, or floating-point values	Operate on the whole string, parts, increment/decrement the integers and floats
LIST	Linked list of strings	Push or pop items from both ends, trim based on offsets, read individual or multiple items, find or remove items by value
SET	Unordered collection of unique strings	Add, fetch, or remove individual items, check membership, intersect, union, difference, fetch random items
HASH	Unordered hash table of keys to values	Add, fetch, or remove individual items, fetch the whole hash
ZSET (sorted set)	Ordered mapping of string members to floating-point scores, ordered by score	Add, fetch, or remove individual values, fetch items based on score ranges or member value



# Команды для работы со строками

**Table 1.3** Commands used on `STRING` values

Command	What it does
GET	Fetches the data stored at the given key
SET	Sets the value stored at the given key
DEL	Deletes the value stored at the given key (works for all types)

# Пример команд в Redis

If a SET command succeeds, it returns OK, which turns into True on the Python side.

Now get the value stored at the key hello.

If there was a value to delete, DEL returns the number of items that were deleted.

```
$ redis-cli
redis 127.0.0.1:6379> set hello world
OK
redis 127.0.0.1:6379> get hello
"world"
redis 127.0.0.1:6379> del hello
(integer) 1
redis 127.0.0.1:6379> get hello
(nil)
redis 127.0.0.1:6379>
```

Start the redis-cli client up.

Set the key hello to the value world.

It's still world, like we just set it.

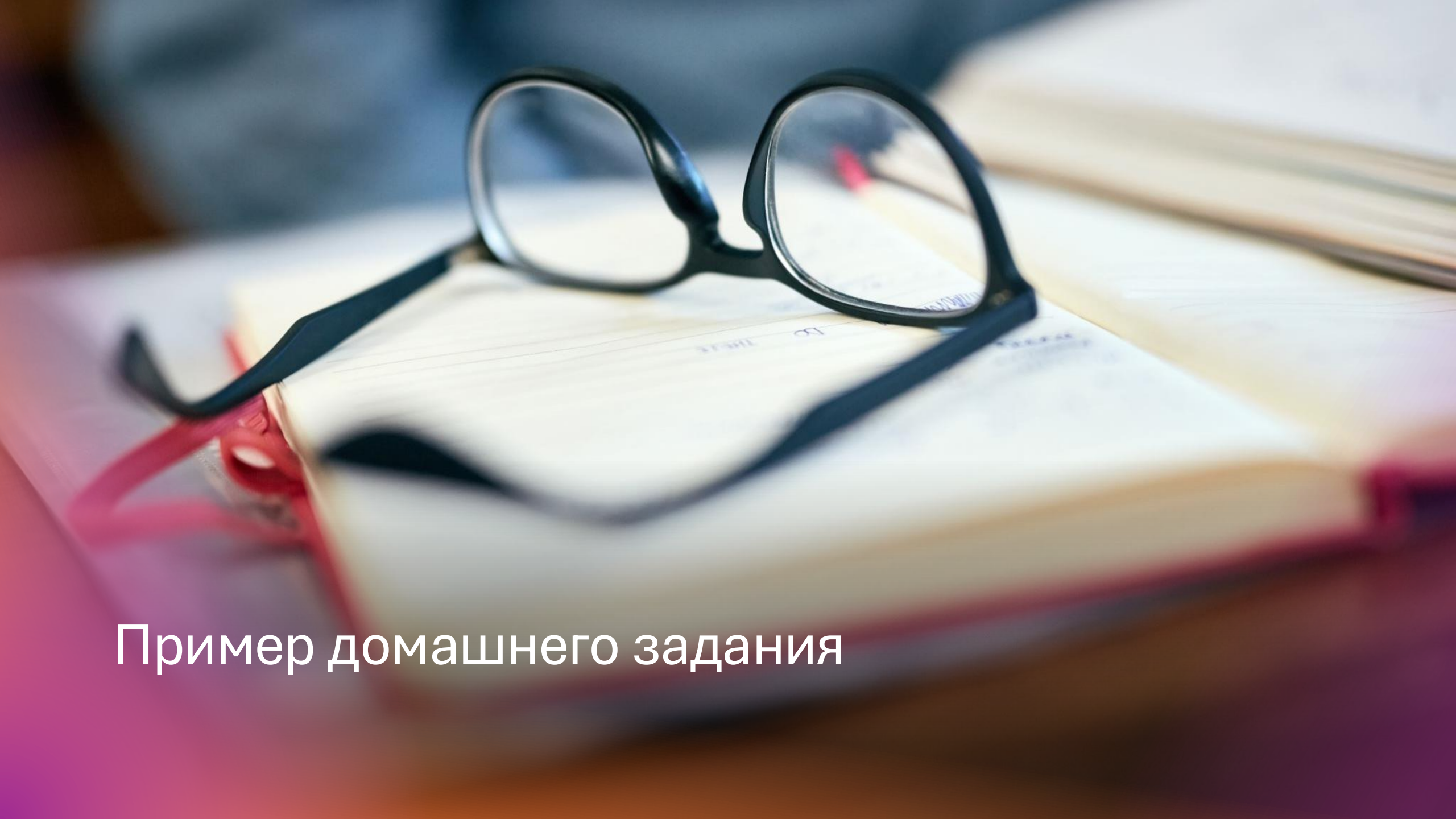
Let's delete the key-value pair.

There's no more value, so trying to fetch the value returns nil, which turns into None on the Python side.

# SET command

```
SET key value [NX | XX] [GET] [EX seconds | PX milliseconds | EXAT unix-time-seconds | PXAT unix-time-milliseconds | KEeptTL]
```

- **EX seconds** -- Set the specified expire time, in seconds.
- *PX milliseconds* -- Set the specified expire time, in milliseconds.
- *EXAT timestamp-seconds* -- Set the specified Unix time at which the key will expire, in seconds.
- *PXAT timestamp-milliseconds* -- Set the specified Unix time at which the key will expire, in milliseconds.
- NX -- Only set the key if it does not already exist.
- XX -- Only set the key if it already exist.
- KEeptTL -- Retain the time to live associated with the key.
- GET -- Return the old string stored at key, or nil if key did not exist. An error is returned and SET aborted if the value stored at key is not a string.



Пример домашнего задания

# Задание

1. Для данных, хранящихся в реляционной базе PostgreSQL реализуйте шаблон сквозное чтение и сквозная запись (Пользователь/Клиент ...);
2. В качестве кеша – используйте Redis
3. Замерьте производительность запросов на чтение данных с и без кеша с использованием утилиты wrk <https://github.com/wg/wrk> изменяя количество потоков из которых производятся запросы (1, 5, 10)
4. Актуализируйте модель архитектуры в Structurizr DSL
5. Ваши сервисы должны запускаться через docker-compose командой docker-compose up (создайте Docker файлы для каждого сервиса)

## Рекомендации по C++

- Используйте фреймворк Poco <https://docs.pocoproject.org/current/>
- Пример по работе с Poco Web Servers и Redis [https://github.com/DVDEmon/arch\\_lecture\\_examples/tree/main/hl\\_mai\\_lab\\_05](https://github.com/DVDEmon/arch_lecture_examples/tree/main/hl_mai_lab_05)

## Рекомендации по Python:

- Используйте FastAPI для построения интерфейсов
- Простой пример применения redis [https://github.com/DVDEmon/architecture\\_python/tree/main/07\\_redis](https://github.com/DVDEmon/architecture_python/tree/main/07_redis)



Что  
почитать?



# Redis

## IN ACTION

Josiah L. Carlson

FOREWORD BY Salvatore Sanfilippo

 MANNING

# На сегодня все

[ddzuba@yandex.ru](mailto:ddzuba@yandex.ru)