بسم الله الرحمن الرحیم

تکنولوژی کامپیوتر

جلسه‌ی دوازدهم
افزایش لود

جلسات گذشته

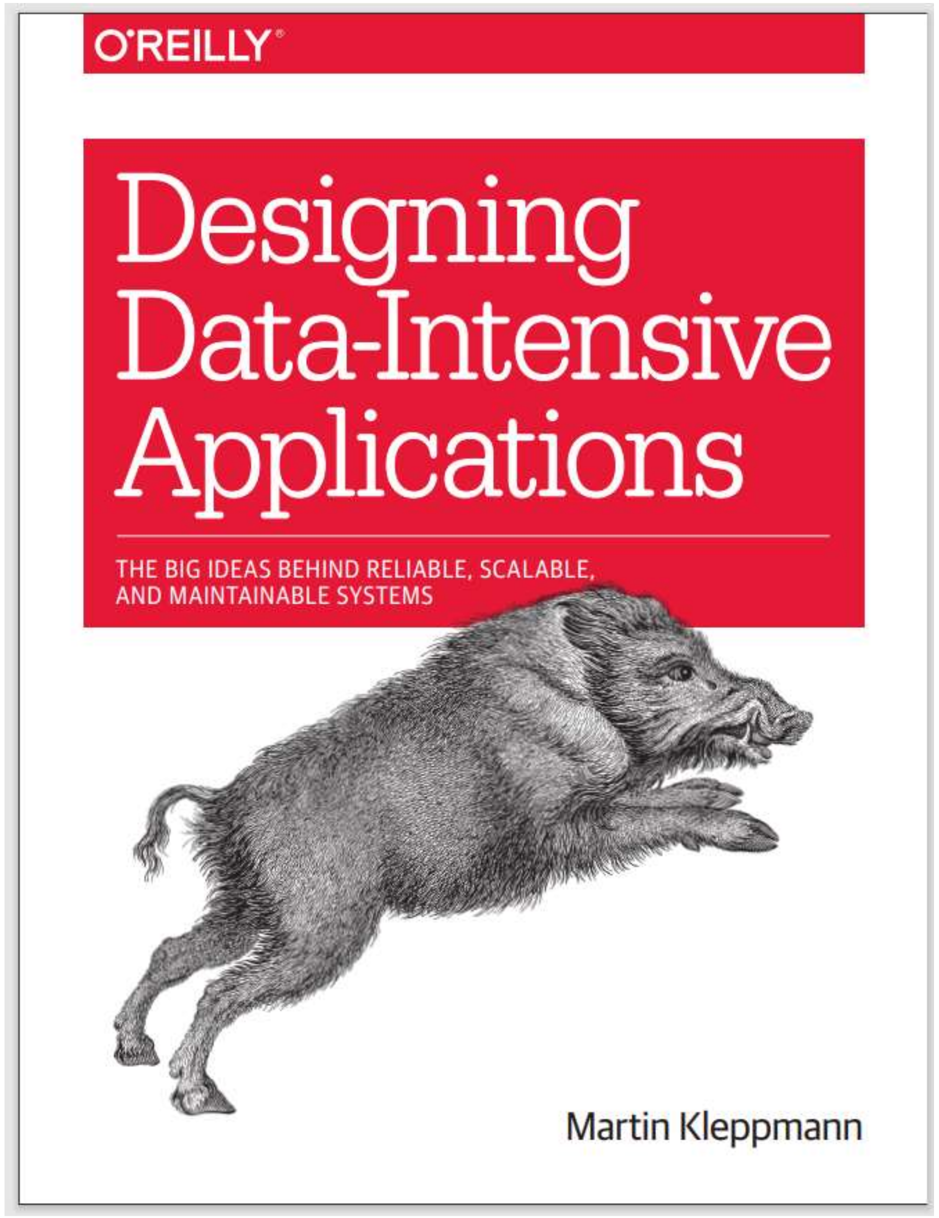
تا اینجا:

- زبان گولنگ رو یاد گرفتیم
- کمی طراحی وب مقدماتی دیدیم
- یک اپلیکیشن روی یک کامپیوتر می‌تونیم بنویسیم
- به یک پایگاه sqlای وصلش کنیم و کار کنیم
- با داکر کار کنیم.

جلسه‌ی جدید

از اینجا به بعد درس

منبع ادامه‌ی درس



O'REILLY

Designing Data-Intensive Applications

THE BIG IDEAS BEHIND RELIABLE, SCALABLE, AND MAINTAINABLE SYSTEMS

Martin Kleppmann

اپلیکیشن‌های داده محور

چرا Data Systems؟

چرا Data Systems؟

مثال‌هایی از اپلیکیشن‌های داده محور

مثال‌هایی از اپلیکیشن‌های Data-Intensive

Commonly needed functions

- Store data so that they, or another application, can find it again later **(databases)**
- Remember the result of an expensive operation, to speed up reads **(caches)**
- Allow users to search data by keyword or filter it in various ways **(search indexes)**
- Send a message to another process, to be handled asynchronously **(stream processing)**
- Periodically crunch a large amount of accumulated data **(batch processing)**

آنچه می‌آموزیم

What we are trying to achieve?

- Reliable
- Scalable
- Maintainable

Data Systems

RELIABILITY

Working correctly

- The application performs the function that the user expected.
- It can tolerate the user making mistakes or using the software in unexpected ways.
- Its performance is good enough for the required use case, under the expected load and data volume.
- The system prevents any unauthorized access and abuse.

faults

- Faults
- fault-tolerant or resilient
- Fault vs failure

Chaos Monkey

Fault: prevention or cure

Hardware Faults

- Hard-disks:
  - *Mean time to failure: 10 – 50 years*
  - *A data-center with 10000 hard disks, one disk failure per day*
- Redundancy to hardware components

Multi-Machine Redundancy

Multi-Machine Redundancy

- In AWS, it is fairly common for VM to become unavailable without warning!
- Reboot single machine to upgrade

Multi-Machine Redundancy

- Load Balancing
- Rolling Upgrade

Load Balancing

Software Faults

- A software bug that causes every instance of an application server to crash when given a particular bad input
- A runaway process that uses up some shared resource
- A service that the system depends on that slows down
- Cascading failures

Solution?

- No quick solution! ☹
- Testing
- Process isolation
- allowing processes to crash and restart
- measuring, monitoring, and analyzing system behavior in production

Human Error

- 50-70% of outages caused by human errors!
- 10-25% of outages caused by hardware fault.

How do we make our systems reliable, in spite of unreliable humans?

- Design systems in a way that minimizes opportunities for error.
- Decouple the places where people make the most mistakes from the places where they can cause failures.
- Test thoroughly at all levels, from unit tests to whole-system integration tests and manual tests
- Allow quick and easy recovery from human errors

How Important Is Reliability?

SCALABILITY

- the system has grown from 10,000 concurrent users to 100,000 concurrent users…
- it is not a one-dimensional label
  - *it is meaningless to say "X is scalable" or "Y doesn't scale."*
  - *"If the system grows in a particular way, what are our options for coping with the growth?"*

Describing Load

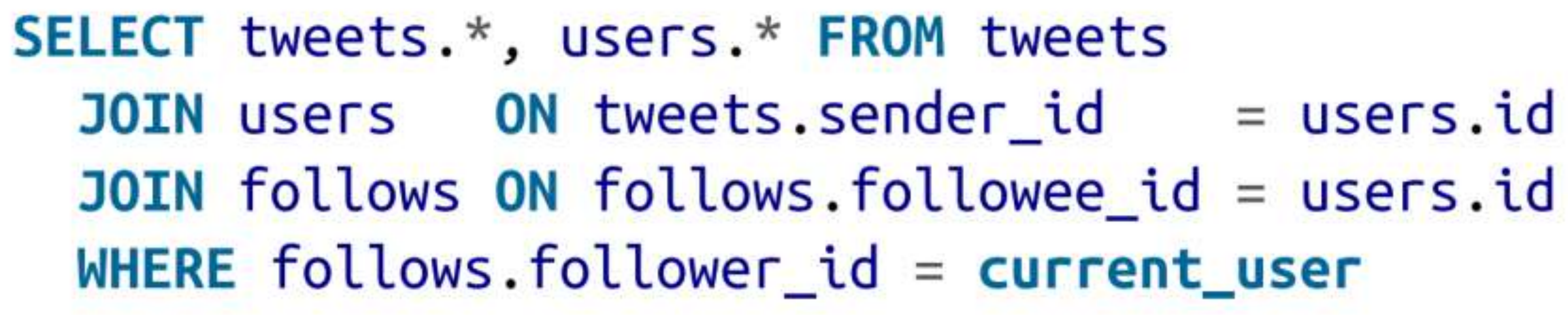
- Load Parameters
  - *Concurrent users*
  - *Requests per second*
  - *Ratio of read / writes of database*
  - *Hit rate of cache*
  - *Or …*

Describing Load - An Example

- Twitter
  - *Post tweet: (4.6k req/sec, 12k req/sec in peak)*
  - *Home timeline: (300k req/sec)*

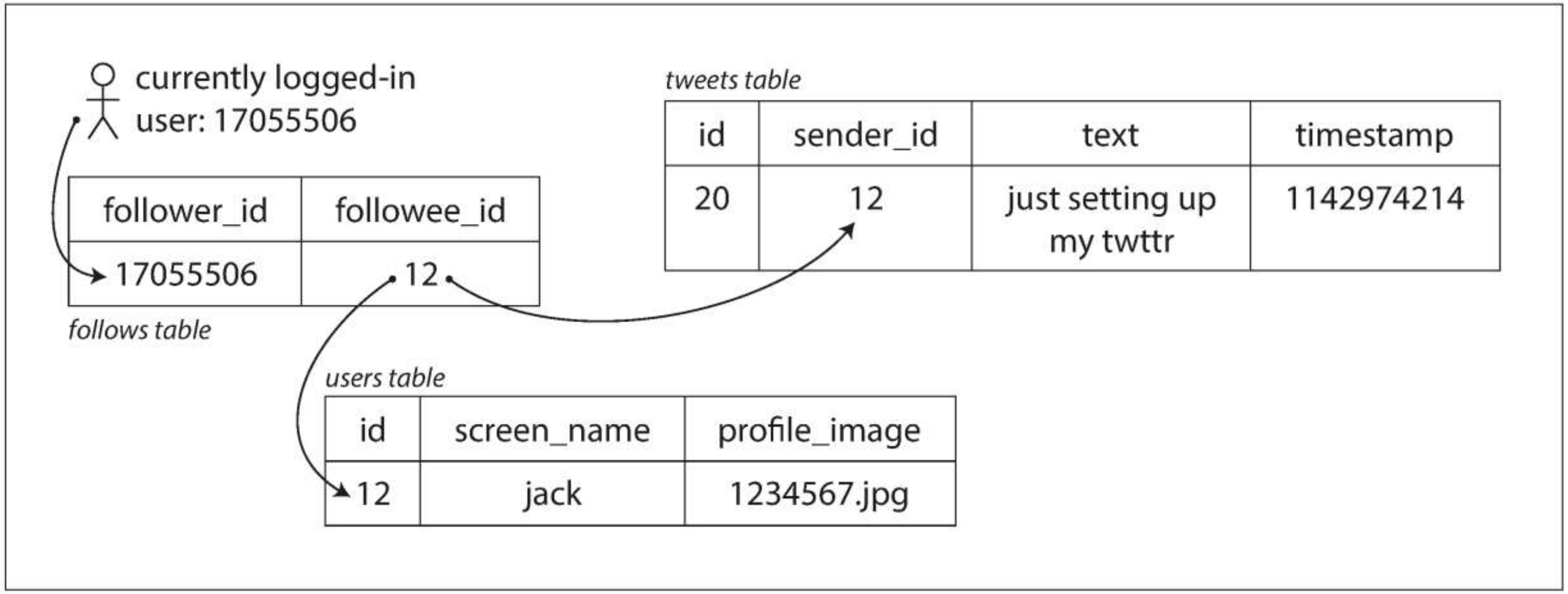
Describing Load - Twitter – Solution 1

- Tweet:
  - *insert tweet on tweets table*
- Home timeline:



```
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
```

Describing Load - Twitter – Solution 1



follows table

follower_id	followee_id
17055506	12

tweets table

id	sender_id	text	timestamp
20	12	just setting up my twttr	1142974214

users table

id	screen_name	profile_image
12	jack	1234567.jpg

Figure 1-2. Simple relational schema for implementing a Twitter home timeline.

Describing Load - Twitter – Solution 2

- Home timeline:
  - *Maintain a cache for each user's home timeline—like a mailbox of tweets for each recipient user*
- Tweet:
  - *look up all the people who follow that user, and insert the new tweet into each of their home timeline cache*
- On average, a tweet is delivered to about 75 followers

Describing Load - Twitter – Solution 2

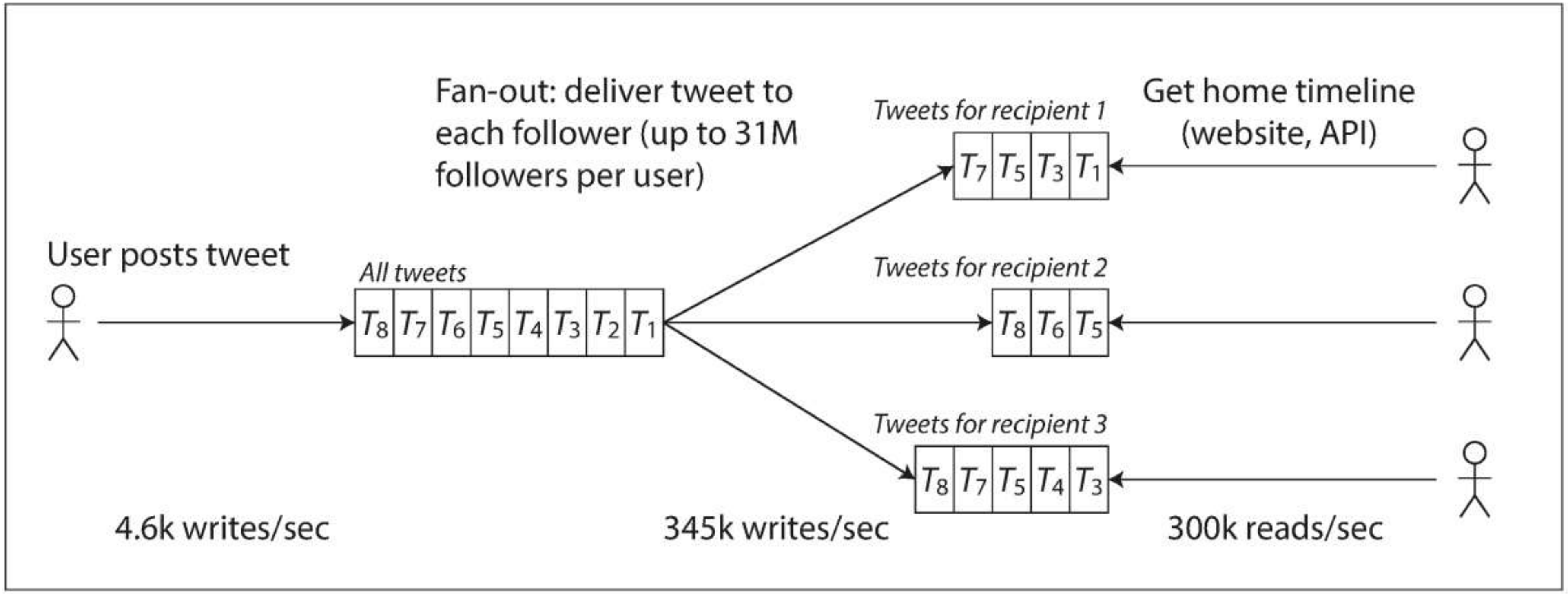

Fan-out: deliver tweet to each follower (up to 31M followers per user)
Tweets for recipient 1
Get home timeline (website, API)
User posts tweet
All tweets
T_8 T_7 T_6 T_5 T_4 T_3 T_2 T_1
T_7 T_5 T_3 T_1
Tweets for recipient 2
T_8 T_6 T_5
Tweets for recipient 3
T_8 T_7 T_5 T_4 T_3
4.6k writes/sec
345k writes/sec
300k reads/sec

Figure 1-3. Twitter's data pipeline for delivering tweets to followers, with load parameters as of November 2012 [16].

- In the example of Twitter:
  - *the distribution of followers per user (maybe weighted by how often those users tweet) is a key load parameter for discussing scalability*

Describing Performance

- Batch Processing:
  - *Throughput: the number of records we can process per second*
- Online systems:
  - *response time: the time between a client sending a request and receiving a response*

Describing Performance

- Average response time?

Describing Performance

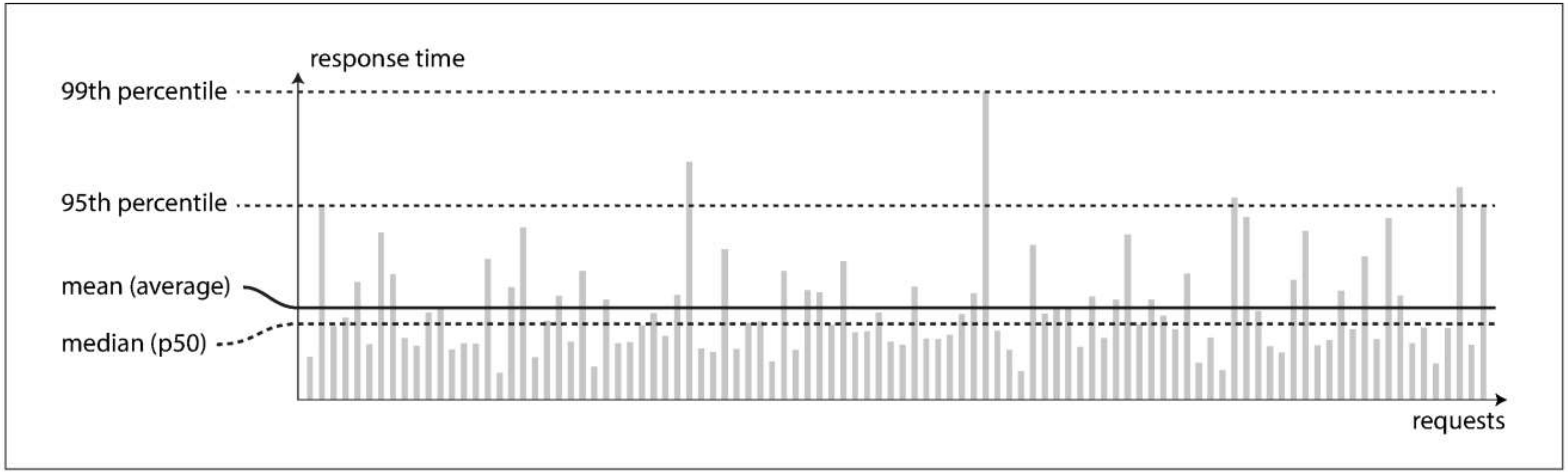


Figure 1-4. Illustrating mean and percentiles: response times for a sample of 100 requests to a service.

Describing Performance

- Percentile

- Amazon case-study

Percentile in Aggregation

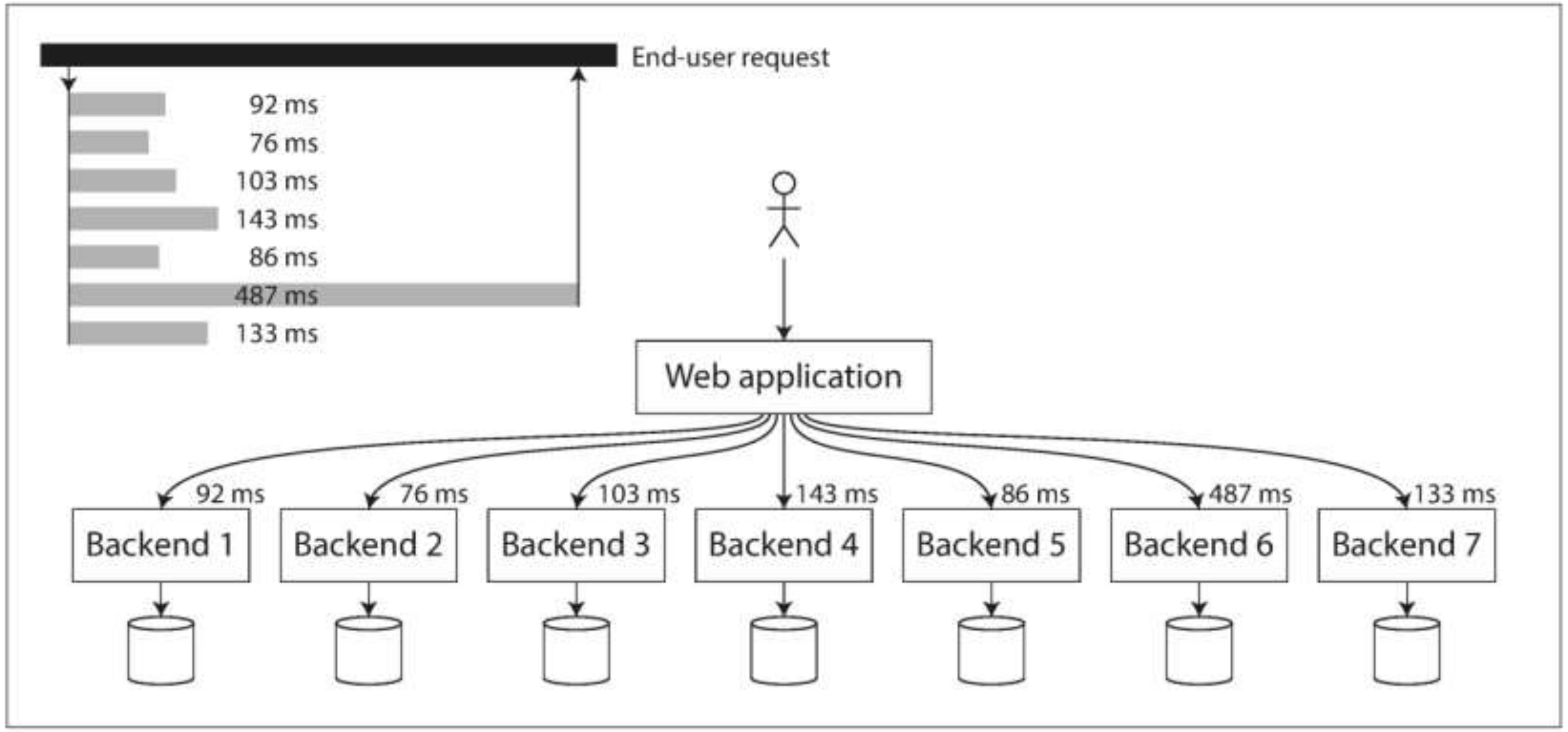


Figure 1-5. When several backend calls are needed to serve a request, it takes just a single slow backend request to slow down the entire end-user request.

Describing Performance

- How to calculate percentile?

Approaches for Coping with Load

- how do we maintain good performance even when our load parameters increase by some amount?

Scaling Up

- vertical scaling, moving to a more powerful machine

Scaling out

- horizontal scaling, distributing the load across multiple smaller machines
- shared-nothing architecture

Scaling?

- Manual Scaling?
- Elastic Scaling?

Handling Large Scale

- Is application specific

MAINTAINABILITY

- majority of the cost of software is not in its initial development, but in its ongoing maintenance—fixing bugs, keeping its systems operational, investigating failures, adapting it to new platforms, modifying it for new use cases, repaying technical debt, and adding new features

Operability

- Make it easy for operations teams to keep the system running smoothly.

Operability

- Monitoring the health of the system and quickly restoring service if it goes into a bad state
- Tracking down the cause of problems, such as system failures or degraded performance
- Keeping software and platforms up to date, including security patches
- Keeping tabs on how different systems affect each other, so that a problematic change can be avoided before it causes damage
- Anticipating future problems and solving them before they occur (e.g., capacity planning)

Operability

- Establishing good practices and tools for deployment, configuration management, and more
- Performing complex maintenance tasks, such as moving an application from one platform to another
- Maintaining the security of the system as configuration changes are made
- Defining processes that make operations predictable and help keep the production environment stable
- Preserving the organization’s knowledge about the system, even as individual people come and go

Simplicity: Managing Complexity

Simplicity: Managing Complexity

- abstraction

Evolvability: Making Change Easy

جلسه‌ی بعد

رپلیکیشن