TDT4225

Chapter 1 – Reliable, Scalable and Maintainable Applictions

Svein Erik Bratsberg
Department of Computer Science (IDI), NTNU

Kunnskap for en bedre verden

NTNU

Technology and tasks

- *Data-intensive* applications as opposed to CPU-intensive
- *Databases*: Store data to be found later
- *Caches*: Remember the result of expensive operations to be read later
- *Search indexes*: Search data by a keyword
- *Stream processing*: Send messages asynchronously to another process
- *Batch processing*: Periodically crunch a large amount of accumulated data

Architecture

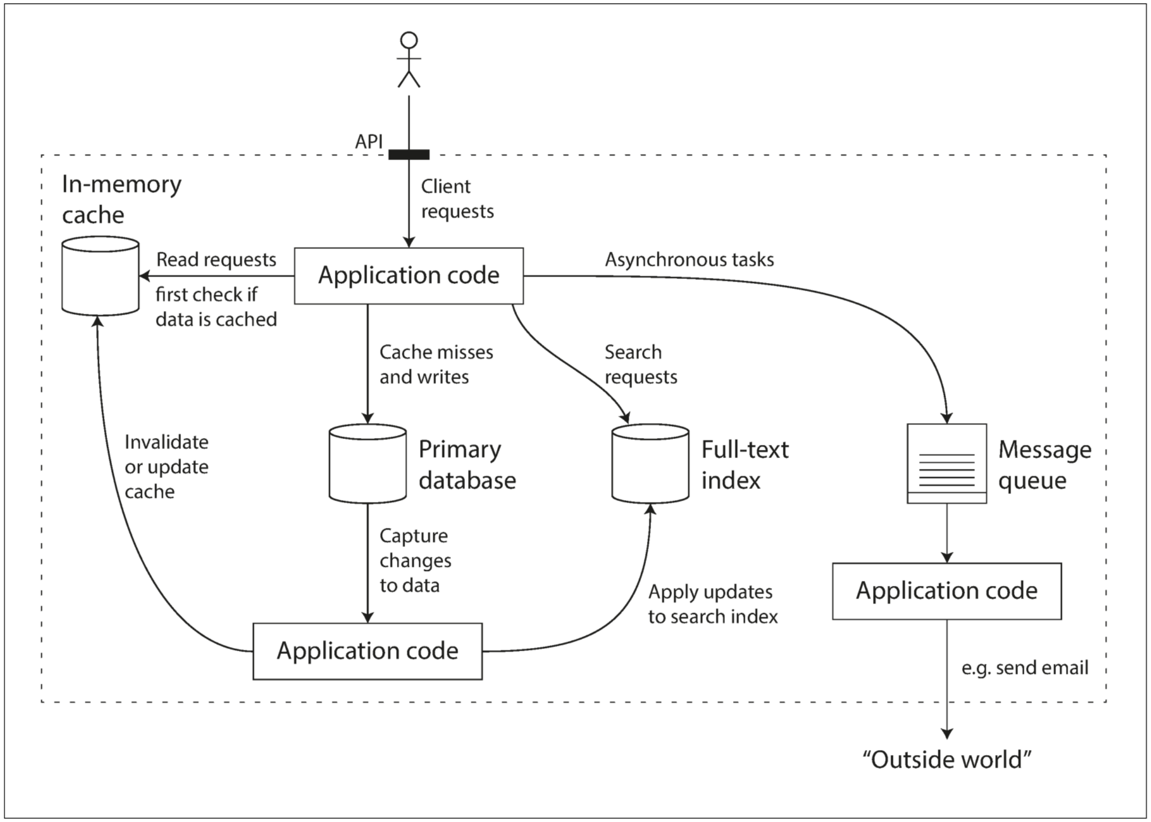

Figure 1-1. One possible architecture for a data system that combines several components.

Three desired properties

- *Reliability*: The system should work correctly even in the face of errors (software / hardware / human)
- *Scalability*: As the system grows (data volume/traffic volume/complexity), it should be dealt with
- *Maintainability*: Different people should be able to maintain and adapt the system over time

Reliability

- *Faults* – the things that can go wrong
- *Fault-tolerant* or *resilient*: Can tolerate certain types of faults
- *Failure*: When the whole system providing service fails
- *Exercise fault tolerance*: Randomly killing individual processes
- Prevention vs. cure of faults.
- Recovery-oriented computing (ROC) vs. replication.

Hardware faults

- Hard disks crash, faulty RAM, power outage, etc
- Hard disks MTTF: 10-50 years. Cluster of 10 000 disks, one disk dies every day
- High availability: Redundancy vs recovery?
- Systems that can tolerate loss of entire machine, as an alternative to redundant hardware. Clusters.

Software errors

- When all computers run the same software, errors may be correlated.
- N-version programming: Different software on each computer?
- Software faults may be dormant for long times, and may be triggered by a special situation?
- Systematic kill and restart may be a solution

Human errors

- Configuration errors by humans the biggest cause of outages
- Hardware faults 10-25 %
- Well-designed APIs and adm. interfaces
- Sandboxing for training using real data without destroying anything
- Testing: Whole system testing, unit testing
- Monitoring performance and errors
- Training and management of systems
- Autonomic computing is appearing slowly

Scalability

- Ability to cope with increased load, users and data
- Load parameters: Requests per second, reads/writes, #active users, etc
- Example twitter (2012)
  - (post) tweet: 12K requests/sec (peak)
  - view tweets: 300K requests/sec
- 1:

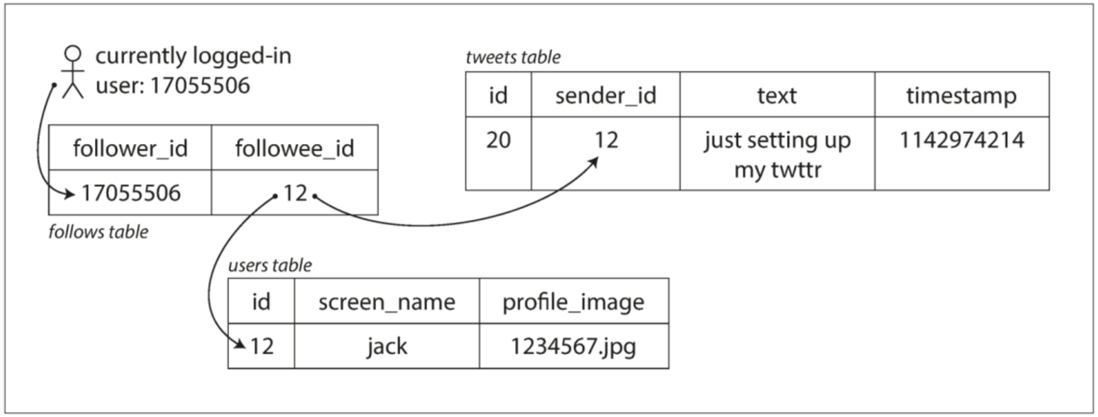

Figure 1-2. Simple relational schema for implementing a Twitter home timeline.

Scalability example - twitter

- 2:

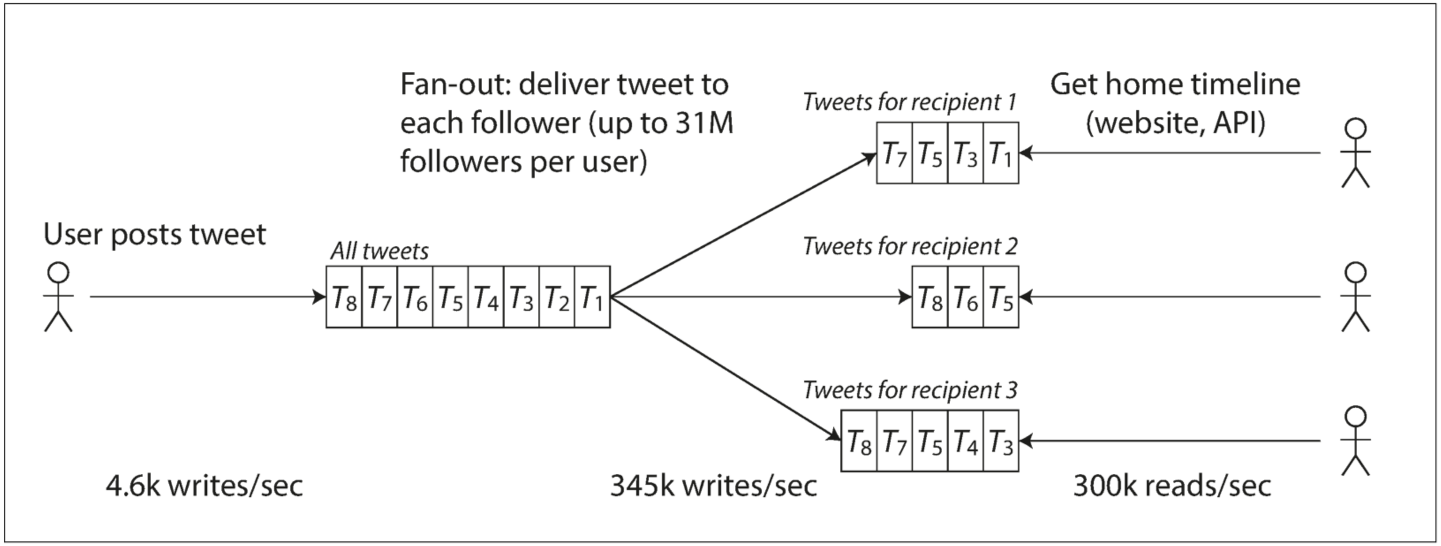

Figure 1-3. Twitter's data pipeline for delivering tweets to followers, with load parameters as of November 2012 [16].

- Hybrid approach used. Tweets from celebrities are handled separately

Performance

- Batch systems: Throughput
- Online systems: Response time
- Distributions: Percentiles vs. average

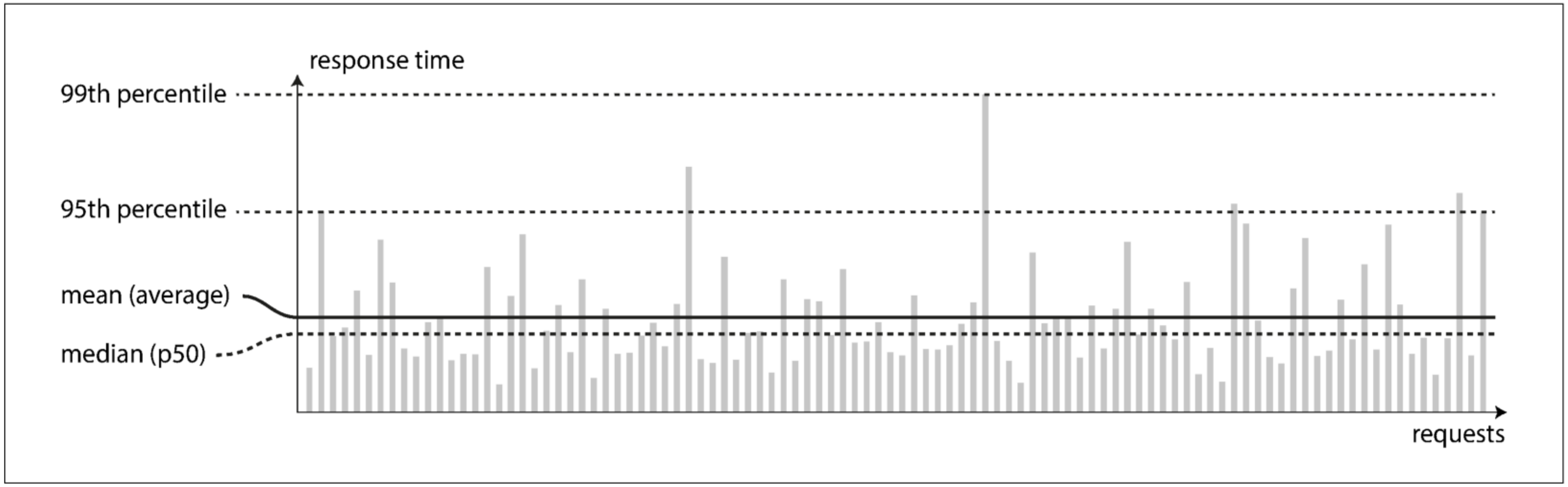

Figure 1-4. Illustrating mean and percentiles: response times for a sample of 100 requests to a service.

Performance and SLA

- SLA – Service level agreement
- Amazon’s SLA measured in the 99.9 percentile (Dynamo paper, 2007)
- Hard due to events outside your control

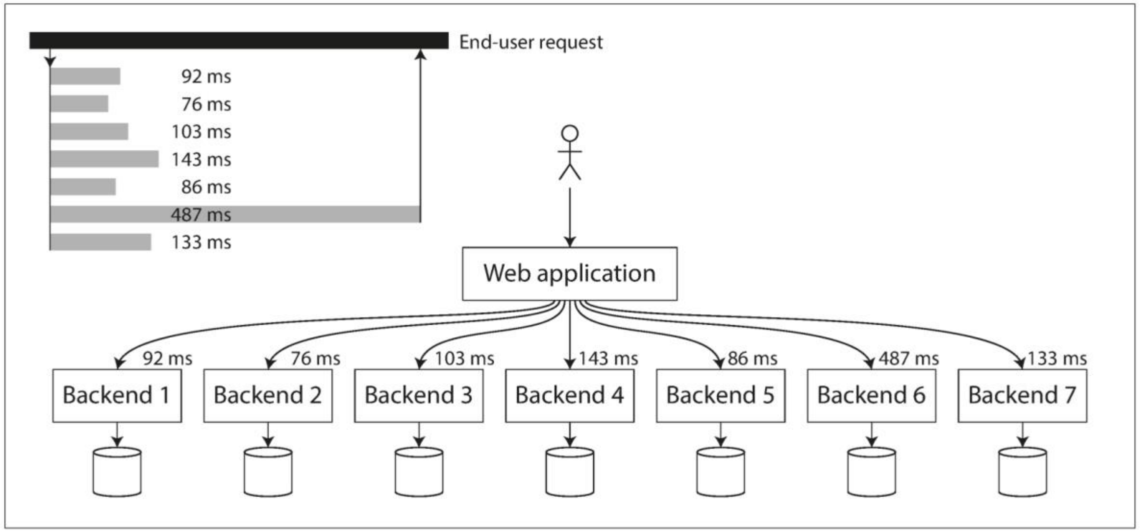

Figure 1-5. When several backend calls are needed to serve a request, it takes just a single slow backend request to slow down the entire end-user request.

Coping with load

- Scaling up (vertical scaling): More powerful machines
- Scaling out (horizontal scaling): More machines to do the work
- Stateless services: Easy to scale out
- Stateful services (e.g. databases): Complex to scale out.
- Sharding/partitioning – resharding/repartitioning

Maintainability

- Major cost of software is in ongoing maintenance, not in initial development
- Operability: Making life easy for operations
- Simplicity: Managing complexity
- Evolvability: Making change easy