T-110.5150
Applications and Services in Internet

Scaling web applications

Denis Shestakov, *fname.lname@aalto.fi*
November 8, 2011

Outline

- What is scalability?
- Scaling strategies
- Scaling network
- Load balancing
- Scaling database
- Monitoring and alerting
- Caching
- Example: eBay architecture (1995-2006)
- *CAP (Brewer's) Theorem* *<- learn more about it!*
- Cloud computing

What is scalability?

- **A service is called scalable if, adding more resources in a system results in increased performance in a manner proportional to resources added**
  - Increased performance? Typically, serving more units of work or handling larger units of work (due to data growth)
- *If resources added to improve redundancy of a system:* a service is scalable, if adding resources does not result in a loss of performance

What is scalability?

- A scalable system has three characteristics:
  - It can accommodate an increased usage
  - It can accommodate an increased dataset
  - It is maintainable
- Scalability is not:
  - Performance: high-performing system may not scale - e.g., it may be fast for 1,000 users and 1 GB data but not that fast with 10 times as many users and 10 times data
  - About using any particular technology: Implementation language has little bearing on the scalability of a system
  - Separation of page logic from business logic

What is scalability?

- *A scalable system has three characteristics:*
  - *It can accommodate an increased usage*
  - *It can accommodate an increased dataset*
  - *It is maintainable*

- Example of a scalable system:
  - *<?php sleep(1); echo "Hello world!"; ?>*
  - Responds after one second: not fast at all, BUT:
    - Traffic growth is accommodated by adding more web servers: no changes in code
    - Dataset growth is also accommodated: no data stored
    - Code is very maintainable: easy to change
  - Note: coded using not the fastest language

Scaling & hardware

- Hardware is usually expensive at the beginning of any project
  - *This is changing now because of cloud computing*
- But after project started, cost of software becomes much more expensive
  - Up to a certain point, where, for huge applications, hardware is an issue again
- Meaning that an application is better to be designed and built in such a way to have little or no software work to scale
- Scale by buying more hardware

Strategies for scaling

- **Vertical** scaling:
  - Get a more powerful version of the same hardware to grow (and throw away the original)
- **Horizontal** scaling:
  - Get an exact duplicate of the current hardware to grow

Vertical scaling

- Start with a basic setup - web server and database server:
  - When each server runs out of capacity, replace it with more powerful
  - When powerful one runs out of capacity, replace it with even more powerful
  - Repeat :)
- Problem:
  - Reach the limit at some point - price grows exponentially
- But:
  - Really easy to design for vertical scaling - no changes in software
  - Fast alternative If the ceiling for application's usage is known

Horizontal scaling

- Start with a basic setup - web server and database server:
  - When each server runs out of capacity, add additional similar
  - When it also runs out of capacity, add additional
  - And so on
- When choosing hardware - consider maintenance costs
  - Rack and cable it, install OS, do basic setup, etc.; other issues: space, power, cooling, etc.
- However, at time goes, dealing with hardware additions and failures can become expensive - in other words, increased administration costs:
  - But, systems administration doesn't scale in cost linearly
- Software performance may not scale linearly:
  - Since it needs to aggregate results from all nodes in a cluster, swap message among all of its peers, etc.
  - At some point, it becomes too expensive to add more hardware

Redundancy

- Machines fail on a regular basis
- One out of every 10,000 machines is expected to die each day
- Have to be prepared to failure of any component
- Spare 'pieces' may be cold, warm and hot:
  - Cold spare: e.g., network switch (physical/software setup and configuration)
  - Warm spare: e.g., queries redirected to a slave database server when a master server dies (configured but needs to be flipped on physically or in software)
  - Hot spare: automatic substitute; e.g., two load balancers are active/passive pair, active takes all traffic and notifies backup balancer via monitoring messages, if active fails, passive stops getting messages and takes over

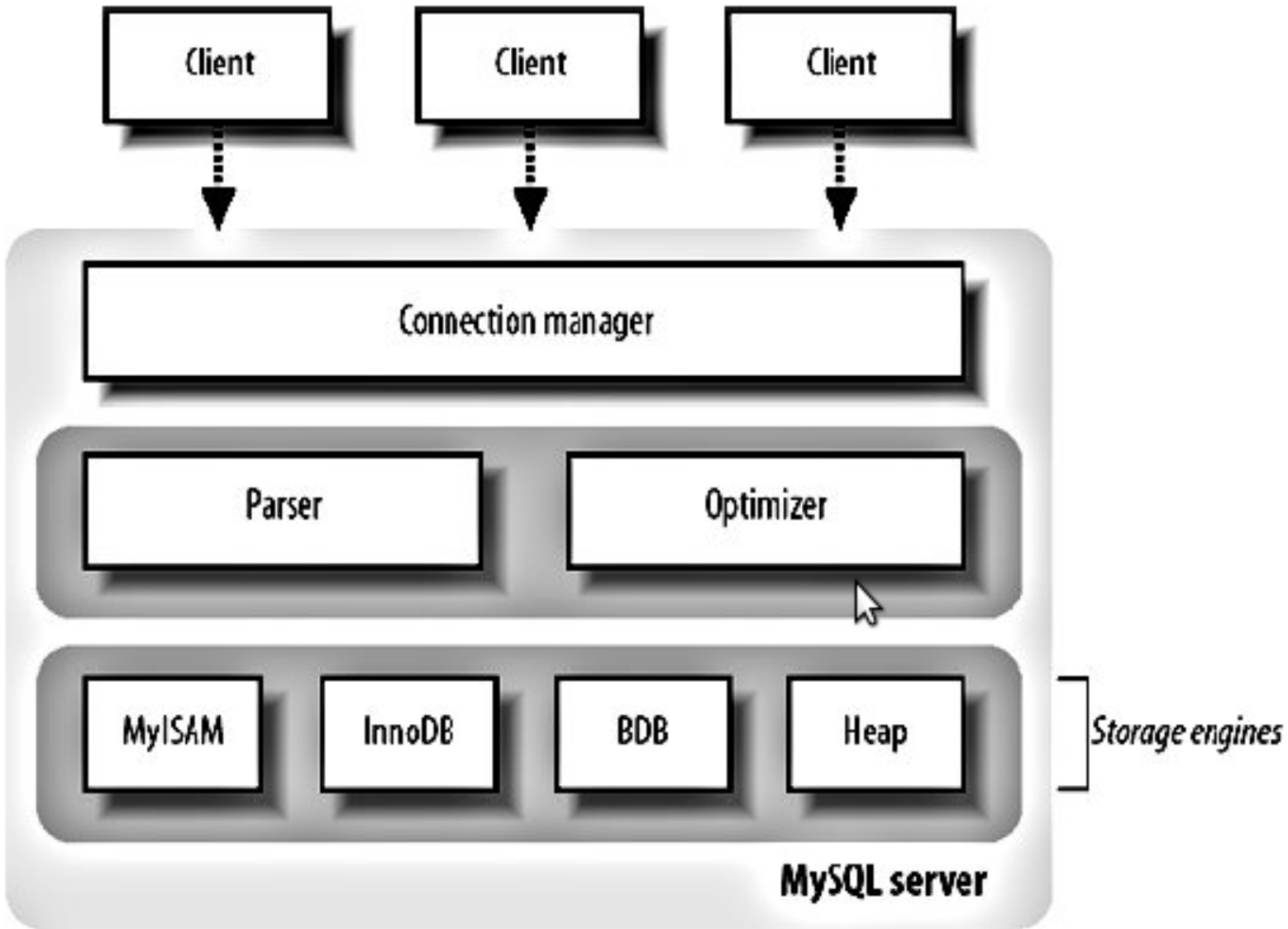
Scaling network

- As a rule, not a big problem
- Regular networking technologies like gigabit Ethernet provide so much bandwidth that web applications most likely never touch the limits
- Sometimes, an application produces a constant stream of noncritical data and occasional bursts of very important data:
  - May be a problem since Ethernet makes no QoS guarantees
  - Split network into distinct subnets
  - Switches support creation of virtual LANs
- When lots of data has to be transferred between two hosts, high-speed data communication can be used: e. g., InfiniBand (data exchange of up to 100 Gb)

Load balancing

- Vertical scaling: spreading load is the job of operating system scheduler
- Horizontal scaling: there are multiple processors, but no operating system to spread requests between them
  - Several solutions grouped under the term "load balancing"
- DNS-load balancing (dns-issues, hard removal)
- Load balancing with hardware (expensive)
- Load balancing with software

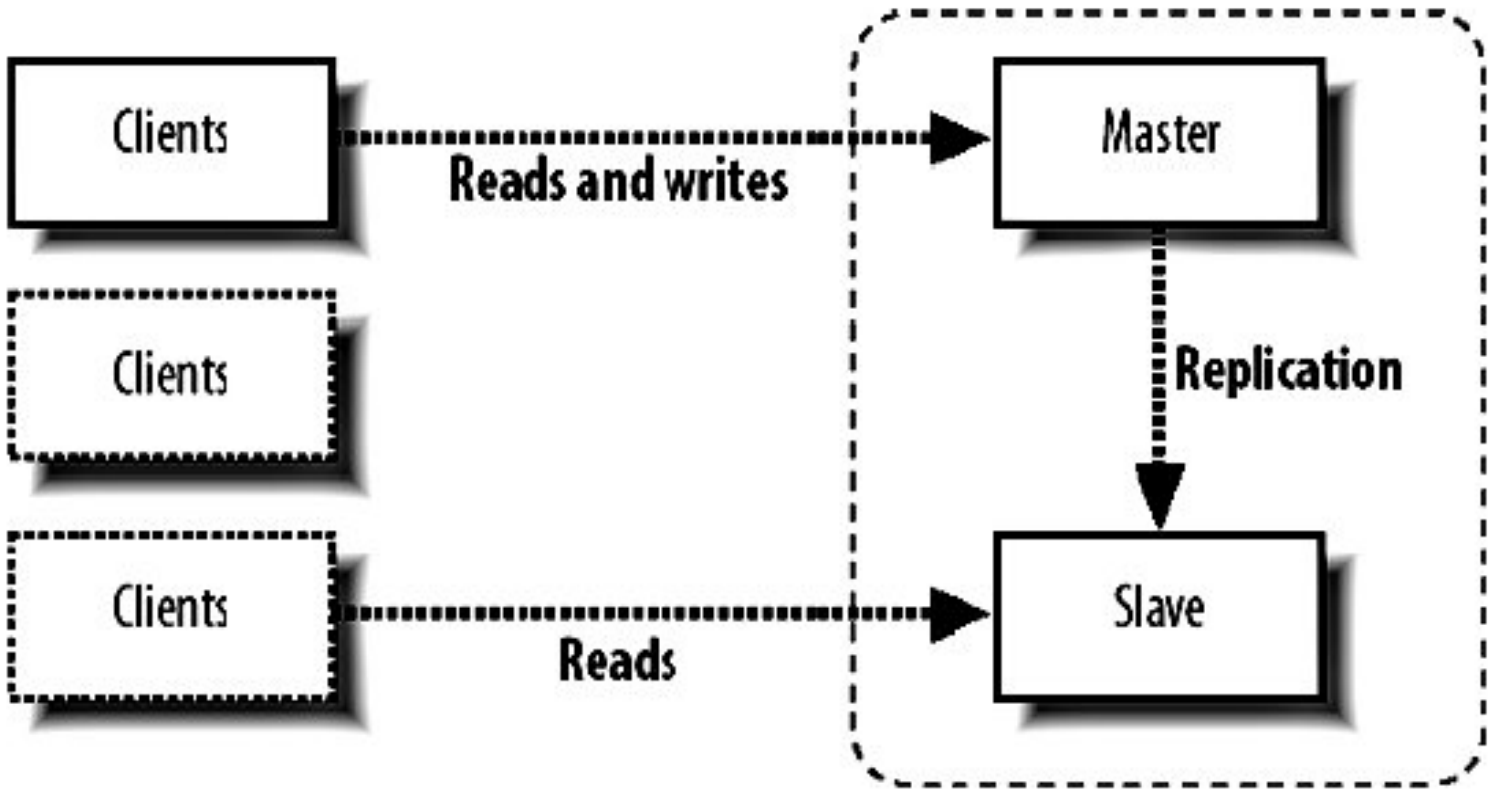
Scaling database



This figure and figures on slides 13-32 have been copied from 'Building scalable web sites', O'Reilly, 978-0596102357, 2006.

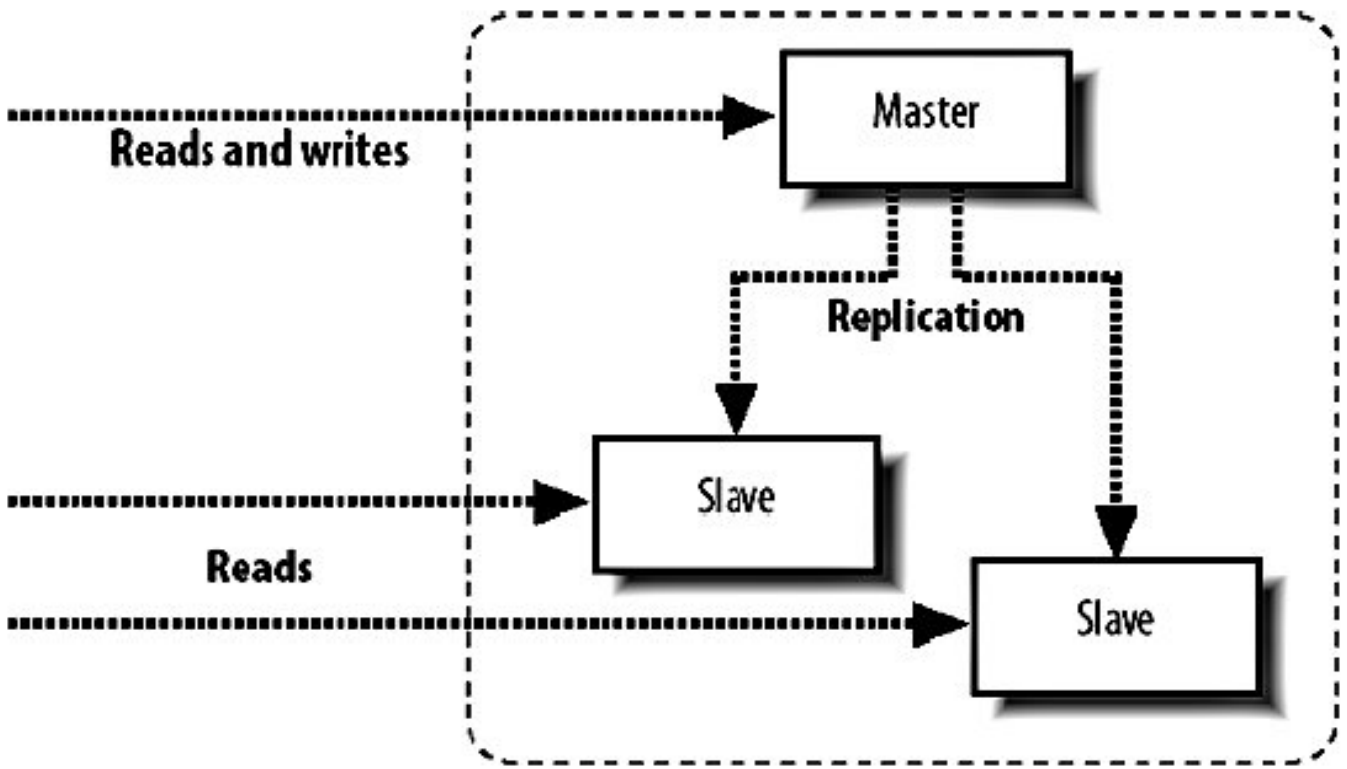
Scaling database

- Web applications typically need a lot more read capacity than write: somewhere between 10 and 100 reads for one write
- Master-Slave replication:



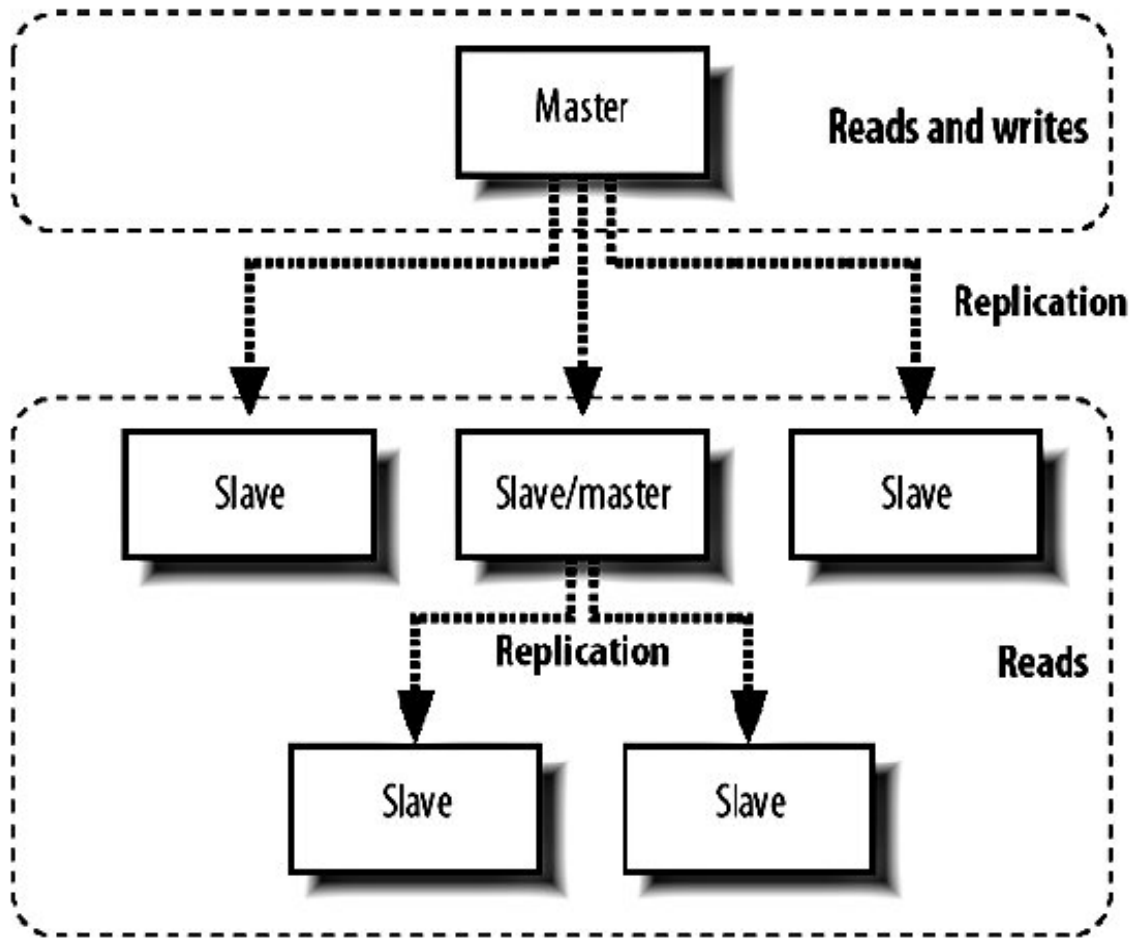
Scaling database

- More read capacity with additional slaves (100 slaves per master in some large applications)
- Slaves aren't guaranteed to be in sync with each other
- Reading scales pretty well, writing is not



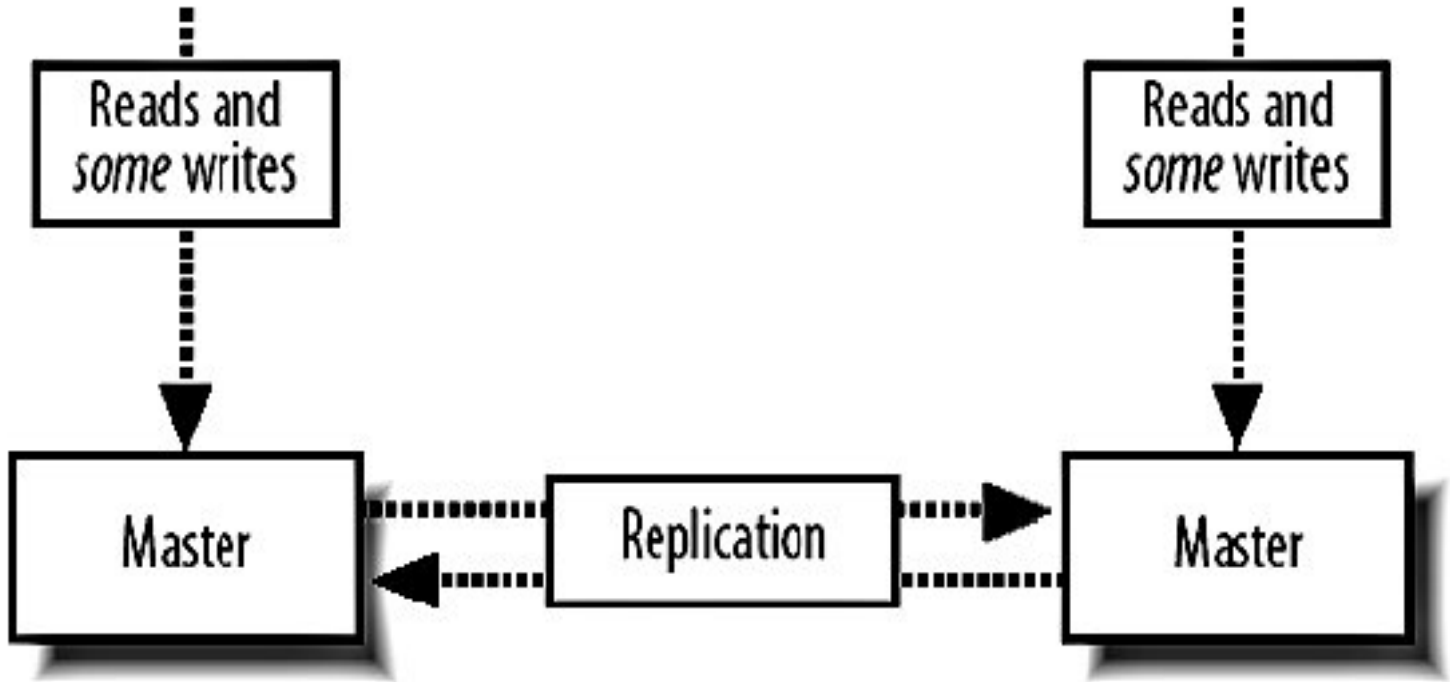
Scaling database

- With lots of slaves, bandwidth required by the master to replicate to all slaves is substantial
- Tree replication



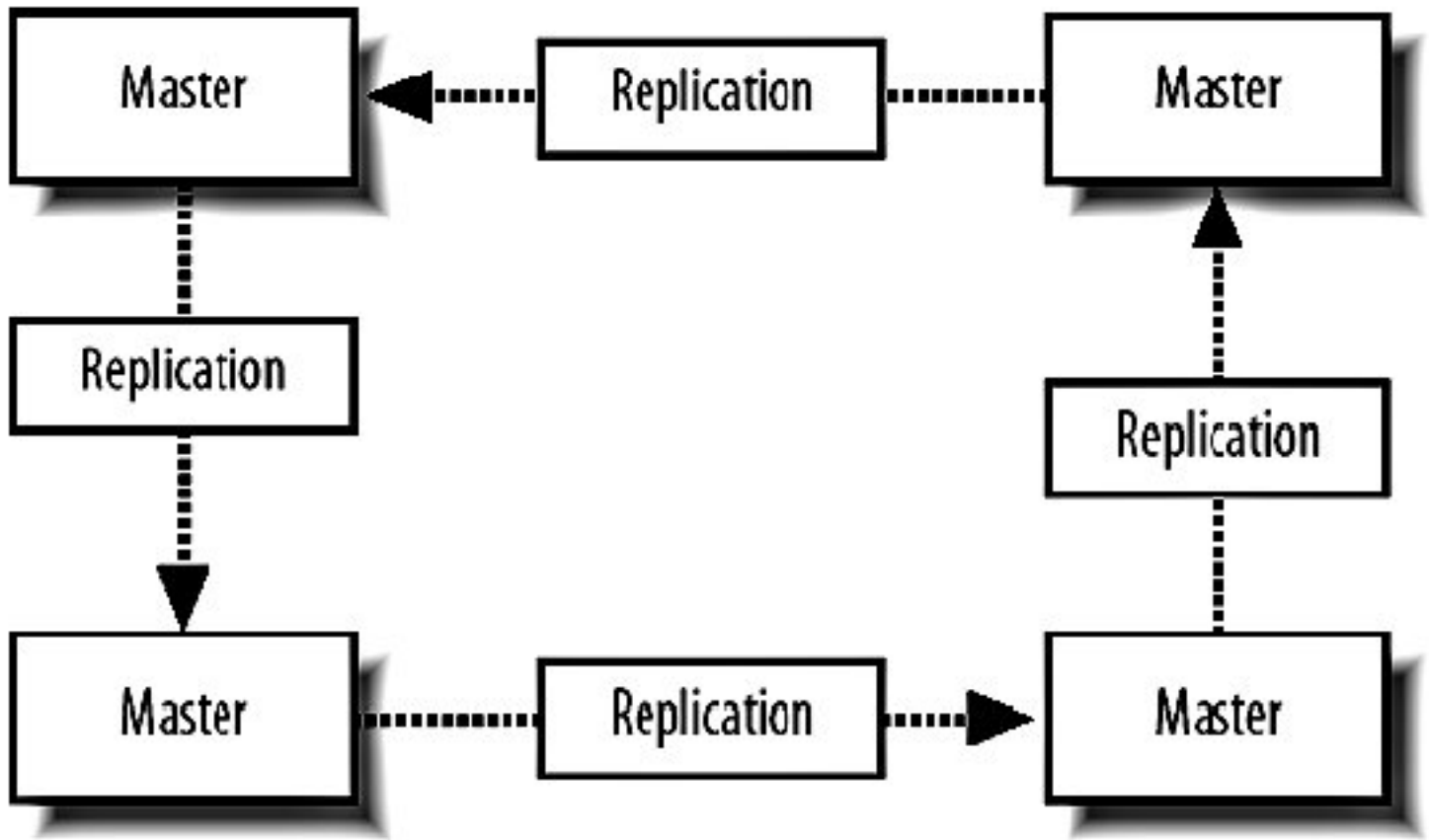
Scaling database

- Master-master replication (each is slave of other)
- Tables with autoincrement primary IDs are problematic



Scaling database

- Masters ring
- At any time, no “true” copy of data
- Failure in a single machine will cause all machines to become stale

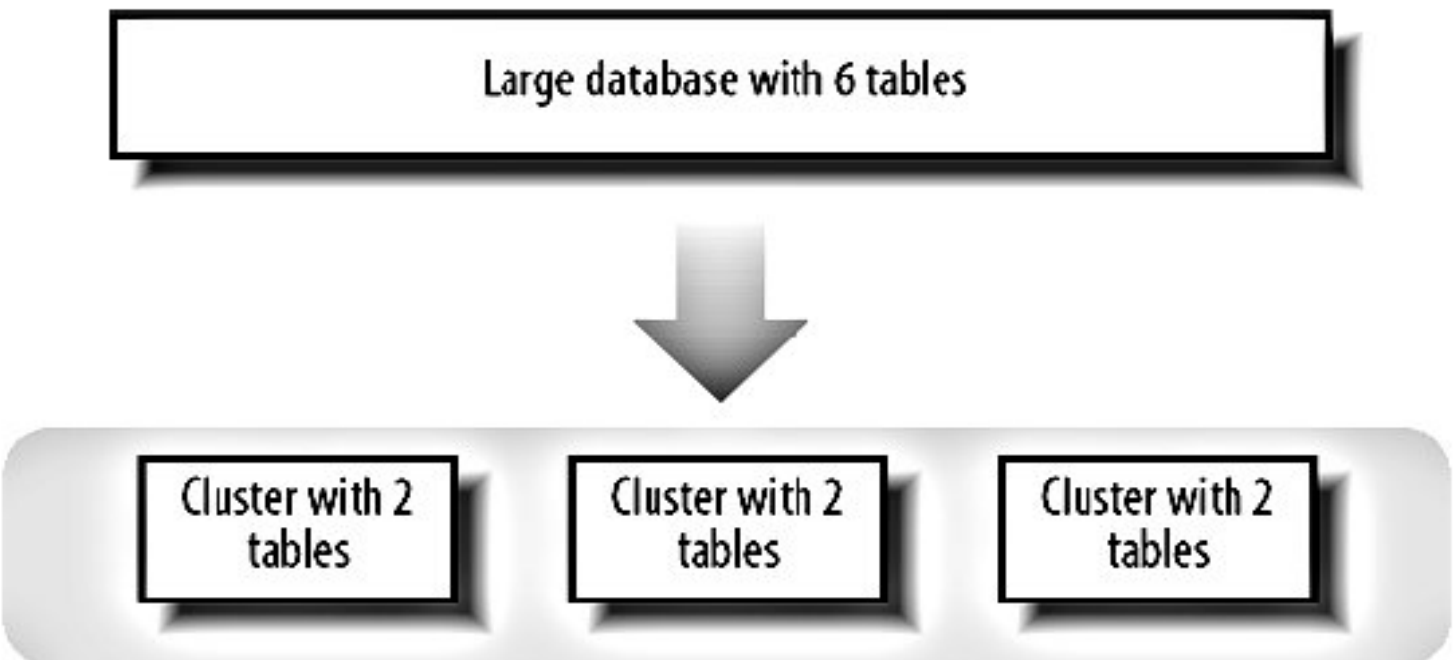


Scaling database

- To allow database to scale writes as well as reads, use database partitioning (chop it up into chunks)
- Two ways:
  - Clustering (vertical)
  - Federation (horizontal)

Scaling database

- Clustering:
  - Each cluster contains a subset of tables (joined tables go together)
- Can split only until a single table or a set of joined tables
- Management of cluster is more difficult (different machines have different data)



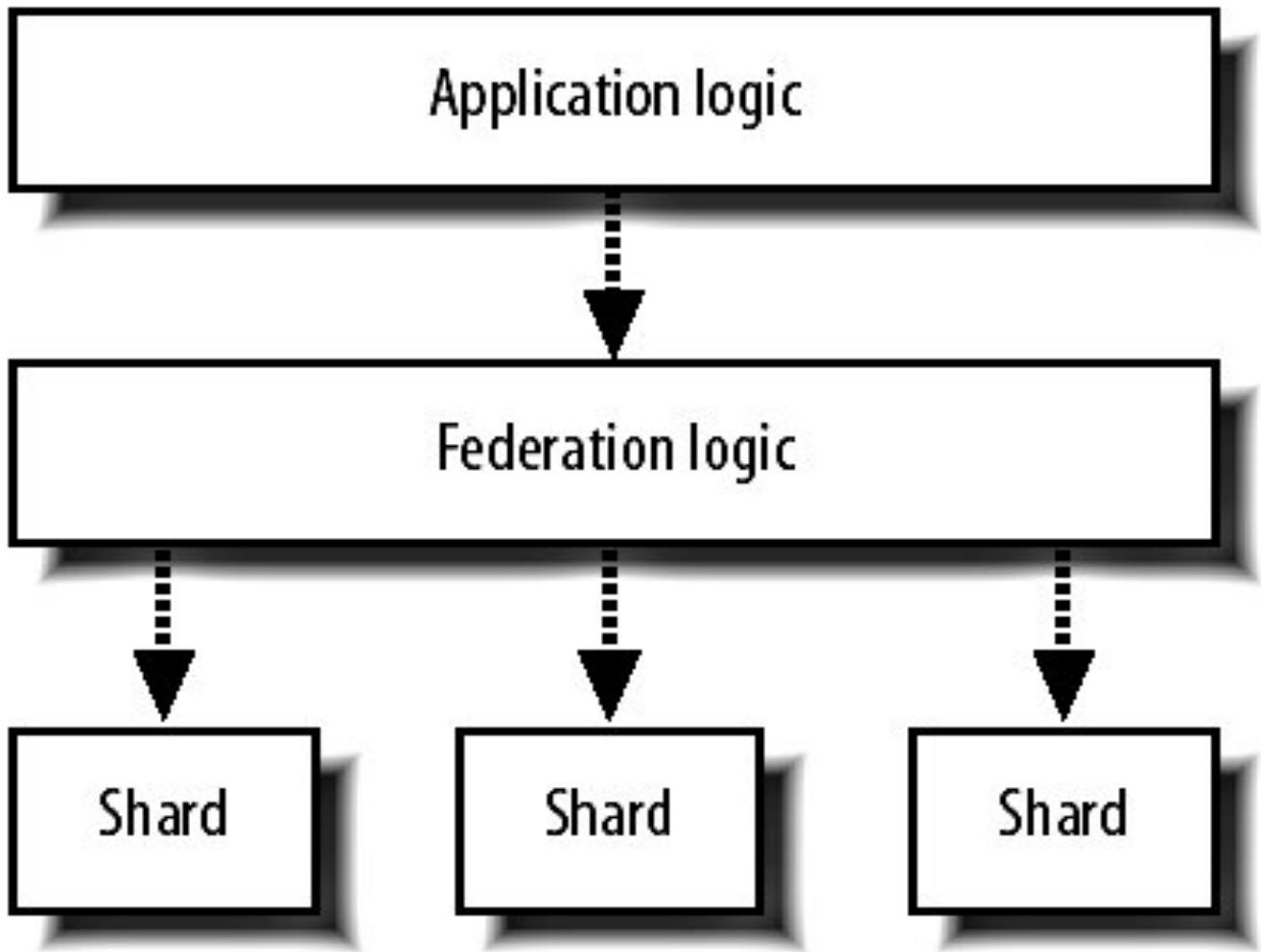
Scaling database

- Federation:
  - Slice the data in the table up into arbitrarily sized chunks
  - Chunks of data and the machines that power them are usually referred to as shards or cells
  - Avoid cross-shard selects and joins
- The key to avoiding cross-shard queries – federate in such as way that all the records you need to fetch together reside on the same shard

- *Note:* horizontal partitioning in MySQL (5.1 and higher): see http://dev.mysql.com/doc/refman/5.1/en/partitioning.html

Scaling database

- Federation logic as a separate layer:



Scaling in brief

- When designing a system to be scalable:
  - Design components that can scale linearly by adding more hardware
  - If you can't scale linearly, figure out the return for each piece of hardware added
  - Load balance requests between clusters of components
  - Take into account redundancy
  - Design your components to be fault-tolerant and easy to recover
  - Federate large datasets into fixed-size chunks
- Your application can only scale as well as the worst component in it:
  - Identifying bottlenecks is a must
  - Monitoring infrastructure required

Monitoring

- In a system consisting of many components something always happens
- Do long-term and extensive monitoring:
  - To understand trends and plan for capacity
- Collect and analyze web logs
- Use beacons (a tiny, (usually) invisible image added to the pages of your application for statistical tracking)
- Stream each logfile to a central location

Application monitoring

- Bandwidth monitoring
- Database statistics
- Server runtime statistics (mod_status for Apache)
- Cache statistics

Alerting

- If something crashes or near to crash, you need to know immediately
- That is, there should be a tool that monitors key statistics and alerts whenever some parameter goes below or above a certain value
- Uptime checks (to check that a service or component is up)
- Threshold checks
- Low-watermark checks

Identifying bottlenecks

- As Donald Knuth said: premature optimization is the root of all evil (or at least most of it) in programming
- Optimizing any part of your application before finding out whether that component requires optimization is a waste of time

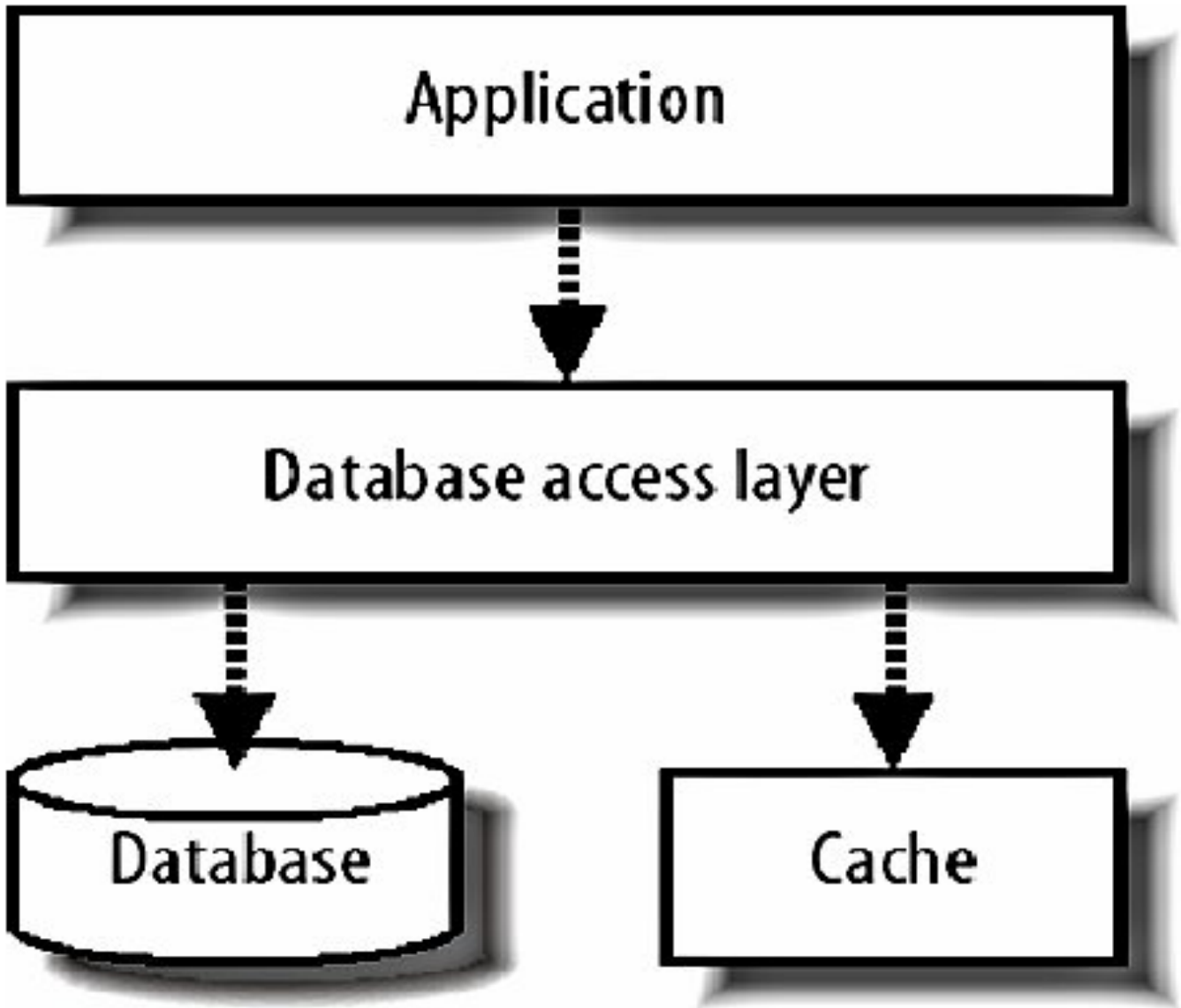
Database throughput

- In many web applications, the biggest bottleneck is database throughput, usually caused by disk I/O
- When a database being a bottleneck, it is generally about the time between a query reaching the database server and its response being sent out
- Certain kinds of data are good for caching:
  - Get set very infrequently but fetched very often
  - E.g., account data: you might want to load account data for every page that you display, but you might only edit account data during 1 in every 100 requests

Adding caching

- When fetching an object, we first check in the cache
  - If object exists, read it from the cache and return it to the requesting process
  - If it doesn't exist in cache, we read the object from the database, store it in cache, and return it to the requester
- When changing an object in the database, either for an update or a delete, it needs to be invalidated in cache

Adding caching

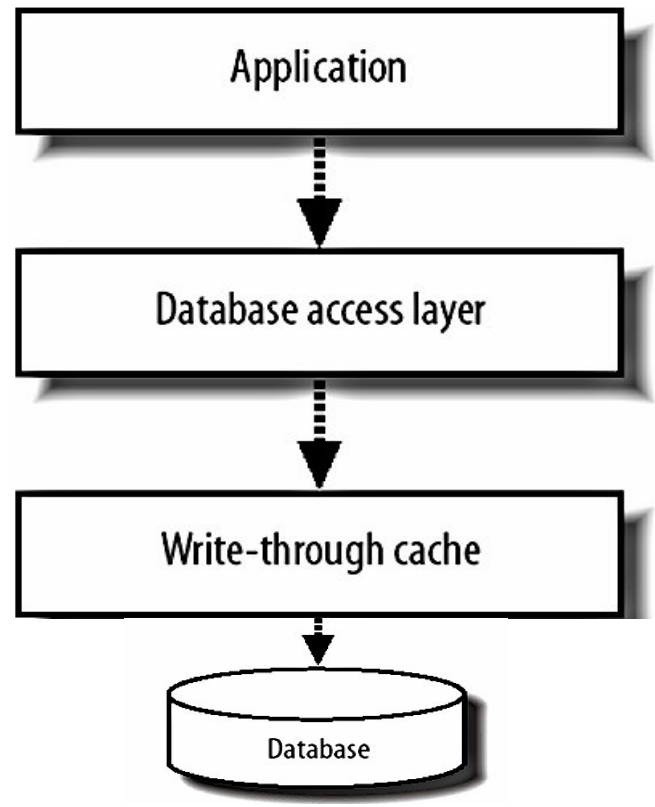


Caching: memcached

- memcached: memory cache demon
- Open source memory-caching system designed to be used in web applications to remove load from database servers
- Supports up to 2 GB of cache per instance
  - But allows you to run multiple instances per machine and spread instances across multiple machines
- Native APIs available for PHP, Perl and other common languages

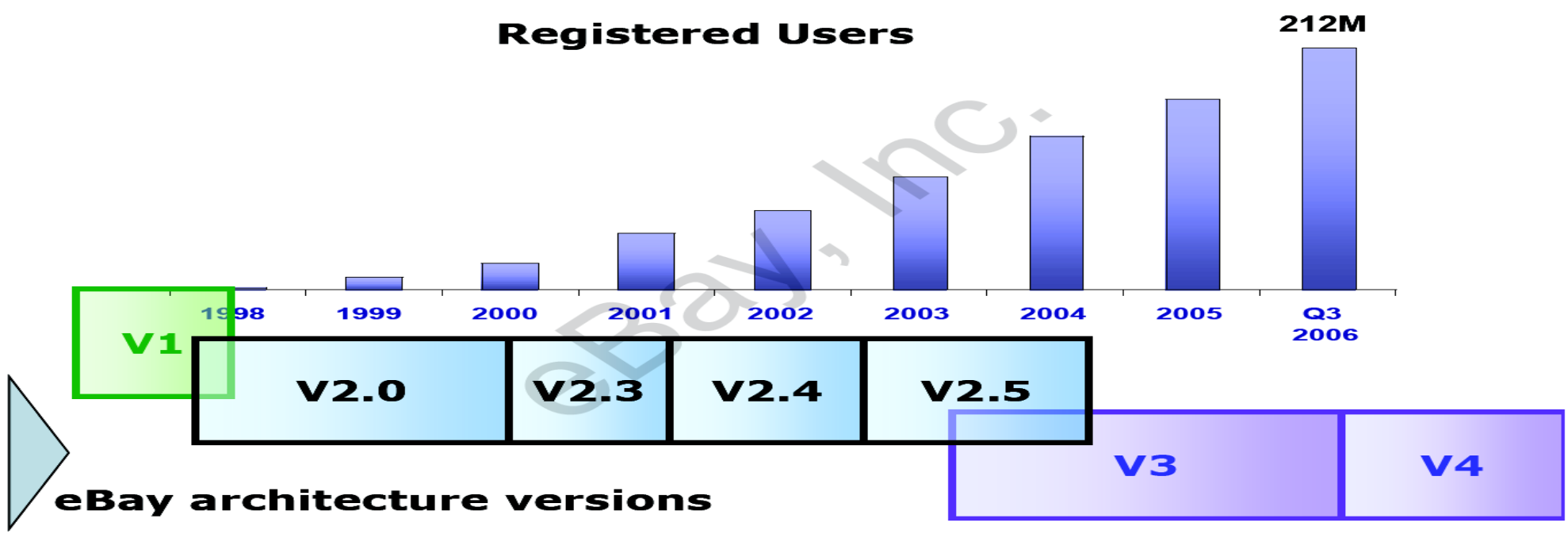
Caching as entire level

- Problems:
  - Reads are caching, while all writes need to be written to database synchronously and cache either purged or updated
  - Cache-purging and updating logic gets tied into our application and adds complexity
- Cache as layer of its own:

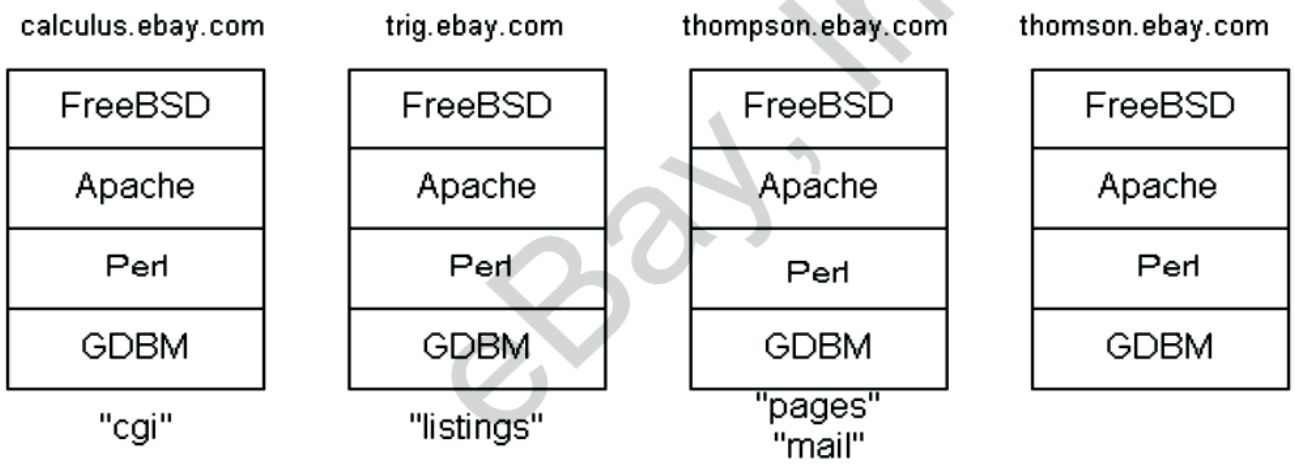


Example: eBay architecture (1995-2006)

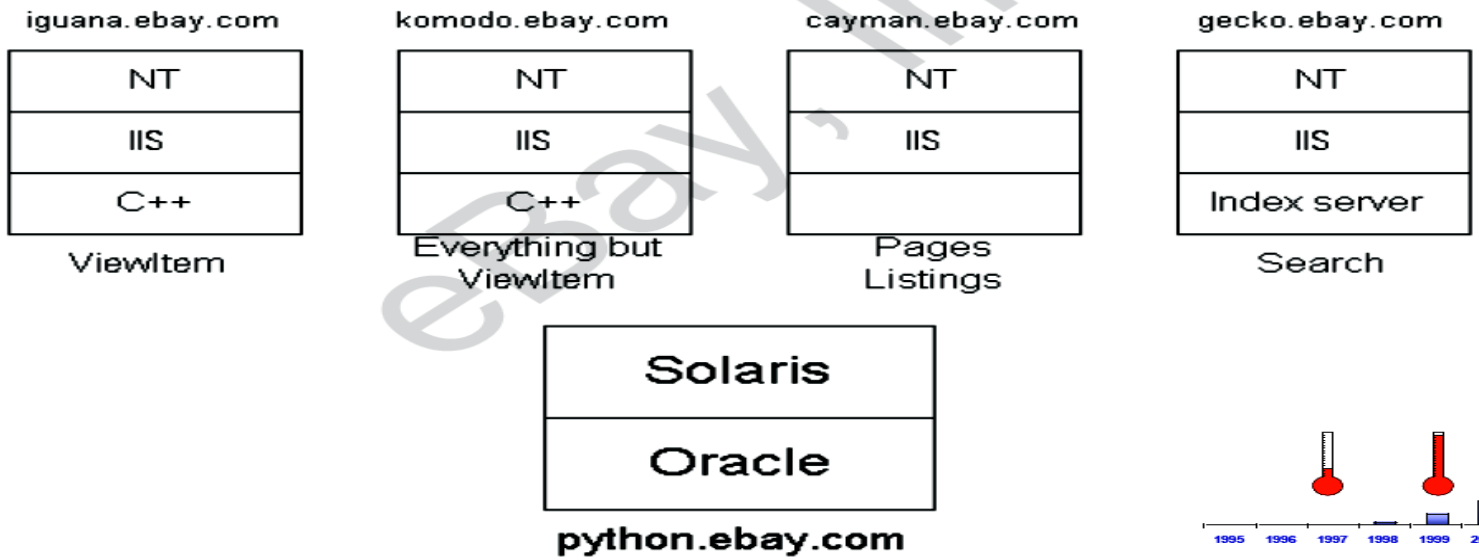
- See http://highscalability.com/blog/2008/5/27/ebay-architecture.html (figures taken from the eBay architecture talk, http://www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf)
- 2006 eBay figures:
  - Over 200mln registered users
  - Over one billion photos
  - Over two petabytes of data
  - Over 26 billions executions per day



Example: eBay architecture

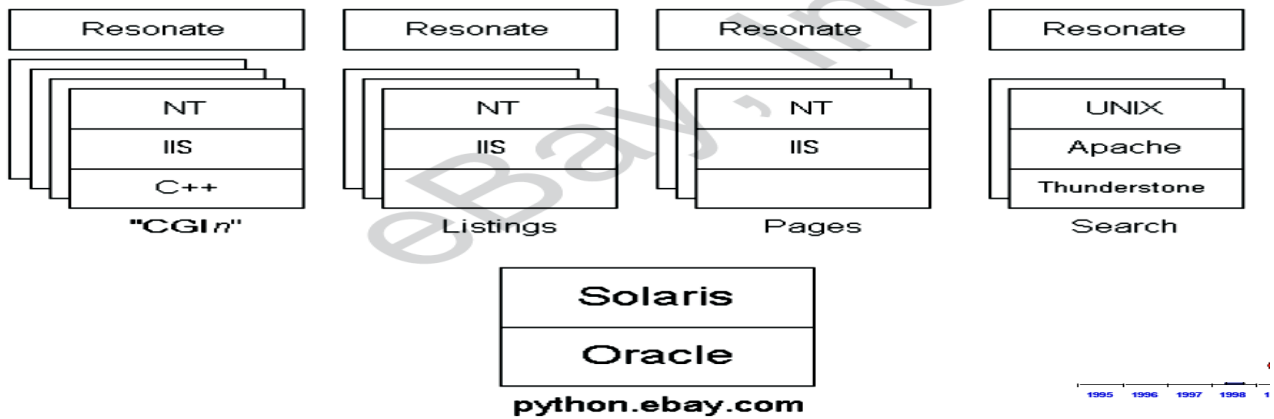
- 1995-97 (v1; every item was a separate file generated by perl script):
  - 

- 1997-99 (v2; C++, MS, Oracle):

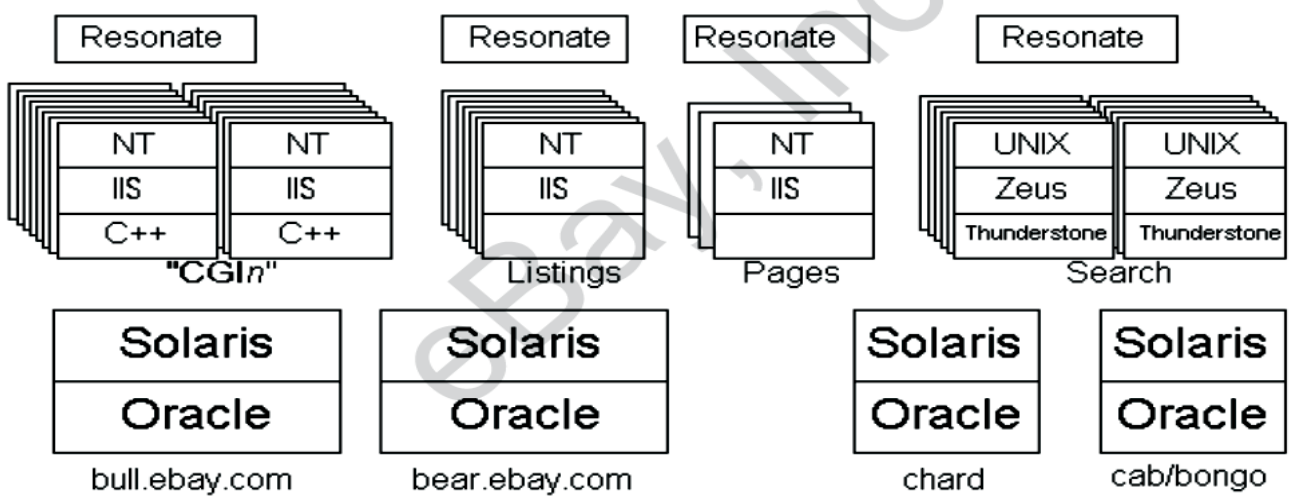


Example: eBay architecture

- 1999 (v2.1; server grouped into pools, front-end balancing and failover, db scaled vertically to a larger machine):



- 1999-2001 (v2.3-2.4; horizontal scaling of servers continued, second db for failover, db 'splitting' started):



Example: eBay architecture

- Observation:
  - Change of technology stack all the time
    - 2002: moved from C++ to Java
    - Later: Oracle->MySQL (sharding)
- Scaling data tier
  - Segmentation by function
    - User hosts, item hosts, account hosts, feedback hosts, ...
    - More that 70 other
  - Horizontal splits within function
  - Introducing logical database hosts (additional layer)

CAP Theorem

- At PODC'00 (in an invited talk "Towards Robust Distributed Systems"), Brewer made the conjecture:
  - It is impossible for a web service to provide the following three guarantees:
    - **C**onsistency
    - **A**vailability
    - **P**artition-tolerance
  - All three are expected and very desirable from real-life web services
- In 2002, Gilbert and Lynch of MIT formally proved the Brewer's conjecture
  - Prove is rather simple

CAP Theorem

- Consistency *(all nodes see the same data)*
  - Formally, atomic consistency
  - Requests to a distributed shared memory are executed like they are executing on a single node (i.e., one by one)
  - Property: read operation that begins after write operation completes must return the value/result of this write operation
- Availability *(node failures do not prevent others to operate)*
  - Every request received by non-failing node must result in a response (or every request must terminate)
- Partition-tolerance *(loss of some messages doesn't disrupt the system operation)*
  - A network is partitioned if all messages sent from nodes in one component of the partition to nodes of other partition are lost

CAP Theorem

- No Consistency
  - web caching, DNS, NoSQL dbs
- No Availability
  - distributed databases, majority protocols
- No partition-tolerance
  - single-site databases, LDAP

Cloud computing

- Servers are not used all the time:
  - Need to have them for redundancy
  - Idle time is cost-inefficient
- Computational power of 1000 servers for one hour costs the same as using power of one server for 1000 hours:
  - Cloud computing services such as Amazon Elastic Compute Cloud made the first option (1000 servers for 1 hour) available
  - One can start with one virtual instance, add more almost instantly when needed *(if, of course, application was designed correspondingly)*

Literature

- Books&articles:
  - *Building scalable web sites*, O'Reilly, 2006
  - *Developing large web applications*, O'Reilly, 2010
  - *The art of capacity planning*, O'Reilly, 2008
  - *Performance by design: Computer capacity planning by example*, Prentice Hall, 2004
  - *Thinking clearly about performance* by Millsap, ACM Queue, 2010
  - *On designing and deploying Internet-scale services* by Hamilton, LISA'07, 2007
- http://highscalability.com/
  - Examples of scaling real web sites
- http://perspectives.mvdirona.com/
- http://en.wikipedia.org/wiki/Scalability
- http://en.wikipedia.org/wiki/CAP_theorem