Scalability Principles

http://www.slideshare.net/jboner/scalability-availability-stability-patterns/48-Buddy_Replication
•Immutability as the default
• Referential Transparency (FP)
• Laziness
• Think about your data:
• Different data need different guarantees

Scalability Trade-offs
Trade-offs
•Performance vs Scalability
•Latency vs Throughput
•Availability vs Consistency

How do I know if I have a performance problem?
If your system is slow for a single user

How do I know if I have a scalability problem?
If your system is fast for a single user but slow under heavy load

You should strive for maximal throughput with acceptable latency

Basically Available
Soft state
Eventually consistent

Availability Patterns
•Fail-over
•Replication
• Master-Slave
• Tree replication
• Master-Master
• Buddy Replication

Active replication - Push
• Passive replication - Pull
• Data not available, read from peer, then store it locally
• Works well with timeout-based caches

Scalability Patterns: State
Partitioning
•HTTP Caching
Reverse Proxy
•RDBMS Sharding
•NOSQL
•Distributed Caching
•Data Grids
•Concurrency

Service of Record
• Relational Databases (RDBMS)

Sharding
•Partitioning
•Replication

When do you need ACID?
• When is Eventually Consistent a better fit?
• Different kinds of data has different needs

Scaling reads to a RDBMS is hard
Scaling writes to a RDBMS is impossible
• NOSQL Databases

NOSQL in the wild
Google: Bigtable
• Amazon: Dynamo
• Amazon: SimpleDB
• Yahoo: HBase
• Facebook: Cassandra
• LinkedIn: Voldemort

Chord & Pastry
Distributed Hash Tables (DHT)
• Scalable
• Partitioned
• Fault-tolerant
• Decentralized
• Peer to peer
• Popularized
• Node ring
• Consistent Hashing

Node ring with Consistent Hashing
Distributed Caching
Write-through
•Write-behind
• Eviction Policies
•Replication
• Peer-To-Peer (P2P)
Peer-To-Peer
• Decentralized
• No “special” or “blessed” nodes
• Nodes can join and leave as they please

Shared-State Concurrency
• Problems with locks:
• Locks do not compose
• Taking too few locks
• Taking too many locks
• Taking the wrong locks
• Taking locks in the wrong order
• Error recovery is hard
Message-Passing Concurrency

Actors

• Share NOTHING
• Isolated lightweight processes
• Communicates through messages
• Asynchronous and non-blocking
• No shared state
… hence, nothing to synchronize.
• Each actor has a mailbox (message queue)

Easier to reason about
• Raised abstraction level
• Easier to avoid
–Race conditions
–Deadlocks
–Starvation
–Live locks

Dataflow Concurrency
• Declarative
• No observable non-determinism
• Data-driven – threads block until
data is available
• On-demand, lazy
• No difference between:
• Concurrent &
• Sequential code
• Limitations: can't have side-effects
STM: restrictions
All operations in scope of
a transaction:
I Need to be idempotent

Behavior
Event-Driven Architecture
Domain Events
• Event Sourcing
• Command and Query Responsibility
Segregation (CQRS) pattern
• Event Stream Processing
• Messaging
• Enterprise Service Bus
• Actors
• Enterprise Integration Architecture (EIA)

"Domain Events represent the state of entities
at a given time when an important event
occurred and decouple subsystems with event
streams. Domain Events give us clearer, more
expressive models in those cases."

Event Sourcing
Every state change is materialized in an Event
• All Events are sent to an EventProcessor
• EventProcessor stores all events in an Event Log
• System can be reset and Event Log replayed

• No need for ORM, just persist the Events
• Many different EventListeners can be added to
EventProcessor (or listen directly on the Event log)

Command and Query Responsibility Segregation (CQRS) pattern
• All state changes are represented by Domain Events
• Aggregate roots receive Commands and publish Events
• Reporting (query database) is updated as a result of the
published Events
• All Queries from Presentation go directly to Reporting
and the Domain is not involved

Messaging
• Publish-Subscribe
• Point-to-Point
• Store-forward
Durability, event log, auditing etc.
• Request-Reply

Actors
Fire-forget
• Async send
• Fire-And-Receive-Eventually
• Async send + wait on Future for reply

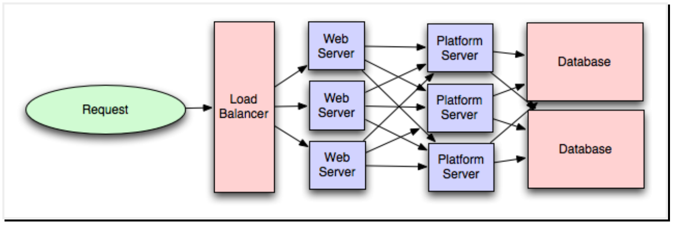
Scability Patterns
•Timeouts
Always use timeouts (if possible):
• Thread.wait(timeout)
• reentrantLock.tryLock
• blockingQueue.poll(timeout, timeUnit)/offer(..)
• futureTask.get(timeout, timeUnit)
• socket.setSoTimeOut(timeout)
•Circuit Breaker
• Let-it-crash
• Embrace failure as a natural state in the life-cycle of the application
• Instead of trying to prevent it; manage it
• Process supervision
• Supervisor hierarchies (from Erlang)
• Fail fast
Avoid "slow responses"
• Separate:
• SystemError - resources not available
• ApplicationError - bad user input etc
• Verify resource availability before starting expensive task
• Input validation immediately
•Bulkheads
Partition and tolerate failure in one part
• Redundancy
• Applies to threads as well:
• One pool for admin tasks to be able to perform tasks even though all threads are blocked
• Steady State
• Clean up after you
• Logging:

• RollingFileAppender (log4j)
• logrotate (Unix)
• Scribe - server for aggregating streaming log data
• Always put logs on separate disk
•Throttling
Maintain a steady pace
• Count requests
• If limit reached, back-off (drop, raise error)
• Queue requests
• Used in for example Staged Event-Driven Architecture (SEDA)

Server-side consistency
N = the number of nodes that store replicas of the data
W = the number of replicas that need to acknowledge the receipt of the update before the update completes
R = the number of replicas that are contacted when a data object is accessed through a read operation

W + R > N strong consistency
W + R <= N eventual consistency

http://lethain.com/introduction-to-architecting-systems-for-scale/
smart client: It is a client which takes a pool of service hosts and balances load across them, detects downed hosts and avoids sending requests their way (they also have to detect recovered hosts, deal with adding new hosts, etc, making them fun to get working decently and a terror to get working correctly).



https://miafish.wordpress.com/2015/03/13/system-scalability-notes/
1. server scale: every server contains exactly the same codebase and does not store any user-related data, like sessions or profile pictures, on local disc or memory.

way to do: this user-related data need to be stored in a centralized data store which is accessible to all your application servers. It can be an external database or an external persistent cache, like Redis

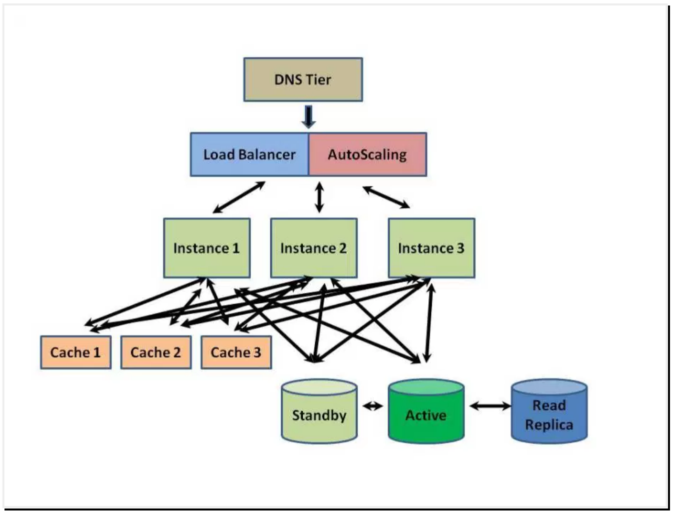
2. Database scale: if there is no much join in query, try NOSql.
3. cache: do not touch database if you can

5. async: implement it as much as you can

Hadoop would be a good desgin

Can't use just one database. Use many databases, partitioned horizontally and vertically.
Because of partitioning, forget about referential integrity or cross-domain JOINs.
Forget about 100% data integrity.
At large scale, cost is a problem: hardware, databases, licenses, storage, power.
Once you're large, spammers and data-scrapers come a-knocking.
Cache!
Use asynchronous flows.
Reporting and analytics are challenging; consider them up-front when designing the system.
Expect the system to fail.
Don't underestimate your growth trajectory.

https://gigadom.wordpress.com/2011/05/04/designing-a-scalable-architecture-for-the-cloud/



http://www.hiredintech.com/system-design/scalability-fundamentals/
Vertical scaling
Horizontal scaling
Caching
Load balancing
Database replication
Database partitioning

Using NoSQL instead of scaling a relational database
Being asynchronous

Vertical Scalability
Horizontal Scalability
Truly stateless components can simply be scaled out and the work load balanced between them
http://www.infoq.com/articles/scalability-principles
1. Decrease processing time

- **Collocation** : reduce any overheads associated with fetching data required for a piece of work, by collocating the data and the code.
- **Caching** : if the data and the code can't be collocated, cache the data to reduce the overhead of fetching it over and over again.
- **Pooling** : reduce the overhead associated with using expensive resources by pooling them.
- **Parallelization** : decrease the time taken to complete a unit of work by decomposing the problem and parallelizing the individual steps.
- **Partitioning** : concentrate related processing as close together as possible, by partitioning the code and collocating related partitions.
- **Remoting** : reduce the amount of time spent accessing remote services by, for example, making the interfaces more coarse-grained. It's also worth remembering that remote vs local is an explicit design decision not a switch and to consider the first law of distributed computing - do not distribute your objects.

2. Partition
this involves breaking up that single piece of the architecture into smaller more manageable chunks. Partitioning that single element into smaller chunks allows you to scale them out

3. Scalability is about concurrency

- If you do need to hold locks (e.g. local objects, database objects, etc), try to hold them for as little time as possible.
- Try to minimize contention of shared resources and try to take any contention off of the critical processing path (e.g. by scheduling work asynchronously).
- Any design for concurrency needs to be done up-front, so that it's well understood which resources can be shared safely and where potential scalability bottlenecks will be.

4. Requirements must be known

5. Test continuously

6. Architect up front

7. Look at the bigger picture

http://highscalability.com/blog/2014/5/12/4-architecture-issues-when-scaling-web-applications-bottlene.html
Capacity Planning

Splitting Database

Database Can Be Split Vertically (Partitioning) Or Horizontally (Sharding).

- Vertically splitting (Partitioning) :– Database can be split into multiple loosely coupled sub-databases based of domain concepts. Eg:– Customer database, Product Database etc. Another way to split database is by moving few columns of an entity to one database and few other columns to another database. Eg:– Customer database , Customer contact Info database, Customer Orders database etc.

- Horizontally splitting (Sharding) :– Database can be horizontally split into multiple database based on some discrete attribute. Eg:– American Customers database, European Customers database.

Architecture Bottlenecks
Scaling bottlenecks are formed due to two issues
Centralised component
A component in application architecture which can not be scaled out adds an upper limit on number of requests that entire architecture or request pipeline can handle.

High latency component
A slow component in request pipeline puts lower limit on the response time of the application. Usual solution to fix this issue is to make high latency components into background jobs or executing them asynchronously with queuing.
http://dev.otto.de/2015/09/30/on-monoliths-and-microservices/

1. ***Vertical decomposition***: The system is cut into multiple verticals that belong entirely to a specific team. Communication between the verticals has to be done in the background, not during the execution of particular user requests.
2. ***RESTful architecture***:
3. ***Shared nothing architecture***: There is no shared mutable state through which the services exchange information or share data. There are no HTTP sessions, no central data storage, as well as no shared code. However, multiple instances of a service may share a database.
4. ***Data governance***: For each data point, there is a single system in charge, a single “Truth”. Other systems have read-only access to the data provider via a REST API, and copy the required data to their own storage.

Data Replication
No Remote Service Calls
http://www.slideshare.net/ssachin7/scalability-design-principles-internal-session
Three goals of application architecture: Scale, HA, performance
Design Principle - Stateless Design
• Stateless designs increases scalability
– Don’t store anything locally on Web Server
• Session State
– Local Sessions – Avoid – Not Scalable
• Load Balancer Sticky sessions can create hot spot load
– Central Session – Good – Distributed Cache, Database
– Client Session – Better – Client Cookie
– No Session – Awesome

Design for Fault Tolerance
– Intent : Enables system to continue its
intended operation, possibly at a
reduced level, rather than failing
completely, when some part of the
system fails
– Drivers: Degraded services are better
than no service at all. Compare cost
effectiveness

• Design for Scaling Out (Bidirectional)
• Stateless Application Design
• Nothing is shared except Database
• Scaling every tier is possible – Web/Service/Database etc.

Design Principle – Loosely Coupled
• Components and layers should be loosely coupled to be able to scale each
layer separately

Caching in Scalability
Design Pattern - Cache Aside Pattern
• Prefer Cache to Database for
Reading
– Intent : Increase read throughput and
reduce database bottleneck
– Drivers: Distributed cache are faster and
shared across web/application servers
– Solution:
• Update cache and database both for
synchronization
• Read from Cache
• Decorator Design Pattern

Design Pattern - Cache Read-through/Write-through (RT/WT)
• Prefer Cache to Database
– Intent: Increase read throughput and reduce database bottleneck. Use
Cache for read write both
– Drivers: Distributed cache are faster and shared across
web/application servers
– Solution:

• Application treats cache as the main data store and reads data from it and writes data to it.
• The cache is responsible for reading and writing this data to the database, thereby relieving the application of this responsibility, asynchronously

Database Scalability
Database Scaling – Replication - Read Mostly Pattern
• Intent: Increase database scalability by separating write and read operations
– Generally most of the applications have around 80% read and 20% write
• Drivers: Separate read write responsibilities, High availability benefits
• Solution:
– Read Write Separation
– Master Slave Pattern

Design Pattern – Partitioning / Sharding
Design Principles – Eventually Consistent
• Basically Available, Soft state, Eventual consistency
• Change in behavior
– Order Placed successfully TO Order Received Successfully

Design Principles – Asynchronous Processing
• Avoid blocking calls, reduce contention
• Queue Based processing Model
• Fire and Forget Calls

Design Principles – Parallel Design
Concern Independent Scaling
• Reliability through Queue
• Queue driven worker tasks - more messages more workers faster work

Queue Based Pattern
Queue - Load Leveling, Load Balancing, Loose Coupling
Design Principles – Queue Based Pattern
• Idempotent
– Design the operation to be idempotent; that is, if it's carried out more than once, it's as if it was carried out just once
– Implement the receiver in such a way that it can receive a message multiple times safely, either through a filter that removes already received messages or by adjustment of message semantics

Design Principles – Capacity Planning
• Everything has a limit: Compose a Scale
– Intent: Design Around Provider SLAs and Capacity
– Solution:
• Know the limits, measure the scalability and increase the scale
• E.g. Storage supports up to 10000 transactions/sec
– Add storage for higher scale
• E.g. Queue supports 5000 messages per seconds
– Add additional Queues (Partitioning) for additional scale

Design Pattern – Multi Site Deployment Pattern

http://www.slideshare.net/techdude/scalable-web-architectures-common-patterns-and-approaches
Custom built
Store last session location in cookie
If we hit a different server, pull our session information across

Remote centralized sessions

But think about queuing
And asynchronous systems

The key to HA is avoiding SPOFs
Some stuff is hard to solve
Fix it further up the tree
Dual DCs solves Router/Switch SPOF

Data federation
Simple things first
Vertical partitioning
Divide tables into sets that never get joined
Split these sets onto different server clusters

Split up large tables, organized by some primary object
Usually users

The wordpress.com approach
Hash users into one of n buckets
Where n is a power of 2

Put all the buckets on one server

When you run out of capacity, split the buckets across two servers
Then you run out of capacity, split the buckets across four servers

Heterogeneous hardware is fine
Just give a larger/smaller proportion of objects depending on hardware

Data federation is how large applications are scaled
Master-master pairs for shards give us HA

Master-master trees work for central cluster (many reads, few writes)

Multi-site HA
Global Server Load Balancing

Dealing with invalidation is tricky
Reverse proxy

HA Storage
RAID is good
RAID 5 is cheap, RAID 10 is fast

Self repairing systems
When something fails, repairing can be a pain

RAID rebuilds by itself, but machine replication doesn't

The big appliances self heal
NetApp, StorEdge, etc

So does MogileFS (reaper)
List of Known Scalable Architecture Templates
LB (Load Balancers) + Shared nothing Units
LB + Stateless Nodes + Scalable Storage
Peer to Peer Architectures (Distributed Hash Table (DHT) and Content Addressable Networks (CAN))
Distributed Queues - This model has found wide adoption through JMS queues
Publish/Subscribe Paradigm
Gossip and Nature-inspired Architectures
Map Reduce/ Data flows
Tree of responsibility - This model breaks the problem recursively and assign to a tree, each parent node delegating work to children nodes. This model is scalable and used within several scalable architectures.
Stream processing - This model is used to process data streams, data that is keep coming. This type of processing is supported through a network for processing nodes. (e.g. Aurora, Twitter Strom, Apache S4)
Scalable Storages

http://www.aosabook.org/en/distsys.html
https://gigadom.wordpress.com/2011/05/04/designing-a-scalable-architecture-for-the-cloud/
The DNS tier – In this tier the user domain is hosted on a DNS service like Ultra DNS or Route 53. These DNS services distribute the DNS lookups geographically. This results in connecting to a DNS Server that is geographically closer to the user thus speeding the DNS lookup times. Moreover since the DNS lookups are distributed geographically it also builds geographic resiliency as far as DNS lookups are concerned
http://highscalability.com/blog/2014/5/12/4-architecture-issues-when-scaling-web-applications-bottlene.html
Response Time Vs Scalability

Response time and Scalability don't aways go together i.e. application might have acceptable response times but can not handle more than certain number of requests or application is handle increasing number of requests but has poor or long response times. We have strike a balance between scalability and response time to get good performance of the application.

Scaling Load Balancer

Load balancers can be scaled out by point DNS to multiple IP addresses and using DNS Round Robin for IP address lookup. Other option is to front another load balancer which distributes load to next level load balancers.

Adding multiple Load balancers is rare as a single box running nginx or HAProxy can handle more than 20K concurrent connections per box compared to web application boxes which can handle few thousand concurrent requests. So a single load balancer box can handle several web application boxes.

Scaling Database
RDBMS

RDBMS database can be scaled by having master-slave mode with read/writes on master database and only reads on slave databases. Master-Slave provides limited scaling of reads beyond which developers has to split the database into multiple databases.

NoSQL
Splitting Database

Database can be split vertically (Partitioning) or horizontally (Sharding).

Vertically splitting (Partitioning) :– Database can be split into multiple loosely coupled sub-databases based of domain concepts. Eg:– Customer database, Product Database etc. Another way to split database is by moving few columns of an entity to one database and few other columns to another database. Eg:– Customer database , Customer contact Info database, Customer Orders database etc.

Horizontally splitting (Sharding) :– Database can be horizontally split into multiple database based on some discrete attribute. Eg:– American Customers database, European Customers database.

Architecture Bottlenecks

Scaling bottlenecks are formed due to two issues

Centralised component A component in application architecture which can not be scaled out adds an upper limit on number of requests that entire architecture or request pipeline can handle.

High latency component A slow component in request pipeline puts lower limit on the response time of the application. Usual solution to fix this issue is to make high latency components into background jobs or executing them asynchronously with queuing.

CPU Bound Application
These issues can be fixed by
Caching precomputing values
Performing the computation in separate background job.

IO Bound Application