CH1 Scale from 0 to millions of users

Key Concepts

- trade-offs of single box solution
- DNS, typical data transmission steps of web app
- Databases
  - SQL vs. NoSQL
  - Vertical vs. Horizontal
  - Shard and challenges
    - resharding data - rapid data growth | uneven distribution
- LB
  - WebApp scale
  - LB strategies
- Data replication
  - read / write separation
  - master - slave architecture
- Caching
  - why use cache?
  - CDN
- Stateful vs. Stateless
  - Stateless - no store user data in web app
- Datacenter
- Message Queue
- Database scaling

Web App

- Data Intensive Application - Computation, Storage, Transmission of Data
  - Scale -> Challenges

Single Server

- Key components

    - User / Client App <--> Web Server
    - DNS - like hash (DN -> IP)
- Mobile vs. Web
    - presentation of data

Database

- Decouple Storage from Business Logic (Computation) - scale independently
- Diagram
    - User / Client <--> Web Server <--> DB
- Database choices
    - Relational DB, structured
        - SQL
        - suitable for data with cleared structure, static schema and types
        - ACID, transactional
        - complex SQL queries
    - NoSQL - Not Only SQL, Schema-less
        - Focus more on scalability across multiple servers
        - limit in ACID
        - large amount of data with simple queries
        - some support complex queries, while others not
        - types
            - K, V
                - Redis / MemCached
            - Document - JSON like objects
                - MongoDB
            - Graph - handle relationship between data points
                - Neo4j - support complex join queries
            - Columnar - optimized for queries over large dataset
                - HBase / Cassandra
        - Join is "typically" not supported
- NoSQL is better fit when requiring
    - super low latency
        - e.g.: k, v store
    - unstructured data, no relational data
    - only need to serialize and de-serialize data
        - serialize: transform data into format easy to store, communicate, cache

        - language agnostic formats

Vertical / Horizontal Scaling

Solve

- when traffic / data scale up, existing server can not handle
  Vertical
- add resources to existing servers
- low traffic scaling
- no redundancy / failover
  Horizontal - add cheap servers to solve vertical limitations:

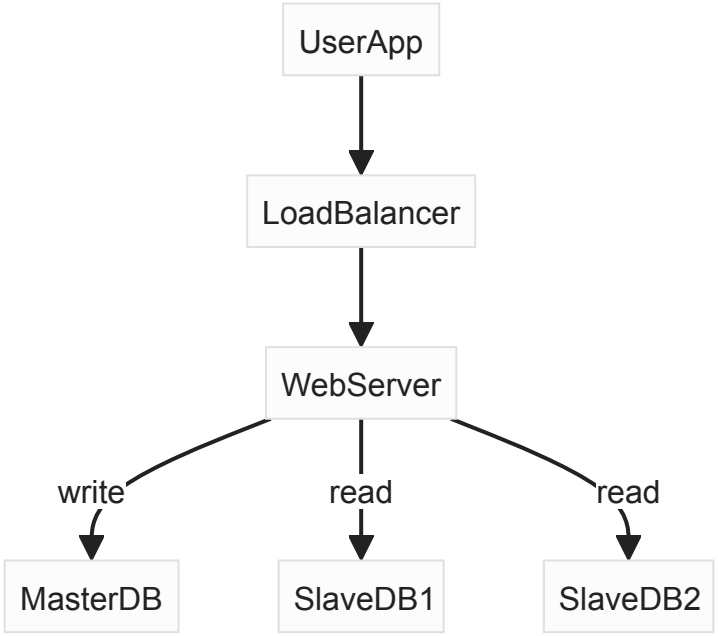
Web Tier Scaling up horizontally - Load Balancer

- App Client <--> LB <--> Servers
- Solve
  - Security - Server is no longer reachable directly by client
  - Evenly distribute incoming traffic
  - No failover concern
- *LB Strategies*
  - Round robin / weighted Round robin
    - best for evenly, stateless process
  - Least Connections
    -
  - Hash
    - best for session persistence
  - Resource based
    - CPU / RAM
  - Application Aware
    - LB can inspect request content and decide distribution
    - slower than Layer-4 LB
- AWS LB
  - ALB - layer 7
    - support features like sticky session
  - NLB - mainly layer 4 - faster than layer 7, less flexible
  - Classic
  - GWLB

        - hash based on fields in the packet headers

Data Tier Scaling up horizontally - DB Replications

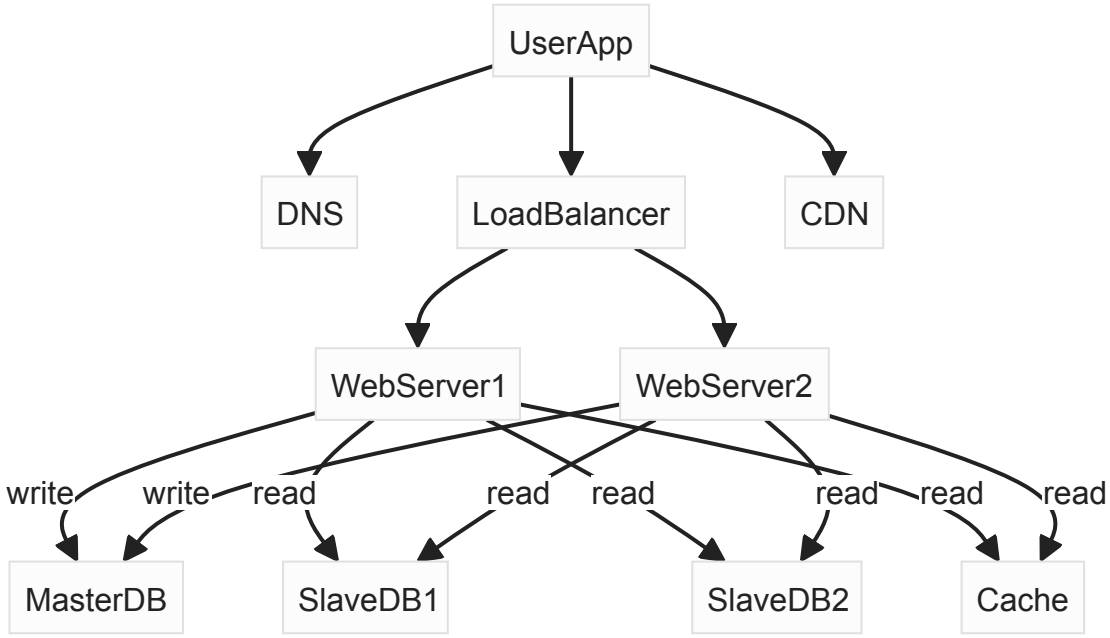
- Solve
    - separate read / write traffic according to use cases
    - good for apps of intensive read, less frequent write
        - e.g.: twitter, blogs, posts
    - failover
- master slave architecture
    - master - only support write
    - slaves - only support reads
- Replicas
    - allows more queries to be served in parallel
    - Reliability - data is preserved under disaster
    - Availability
- If master goes offline
    - slave - promote to master
    - in production - more complex promotion, need to run data recovery scripts



Improve Load/Response Time - Cache Tier

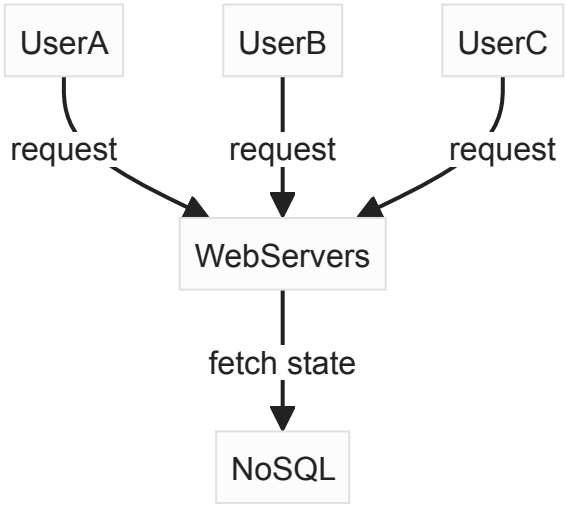
- Read-through
    - read database only when cache miss

    - write cache when get new values
- When use cache, consider:
    - cache is only for read frequently ***but modified infrequently***
        - consist issue with data sources(DB)
    - Expiration policy
    - Consistency
    - Single Cache server represents a potential single point of failure
        - Recommend :overprovision the required memory by certain percentage
    - Eviction Policy:
        - evict when cache is full
        - policies
            - LRU
            - LFU
            - FIFO
- CDN
    - Cache static content near user
    - TTL of content
    - Consider:
        - Cost: cache infrequent assets provides no big benefits
        - Expiry
        - CDN Fallback
        - Invalid files
            - use API to delete
            - use versioning of files, add param to URL, e.g.: `image.png?v=2`



Stateless Architecture

- stateful: remember client data (state) from one request to next
- stateless: keeps no user information
- when new request goes to a server, the server need to match session data, else fail



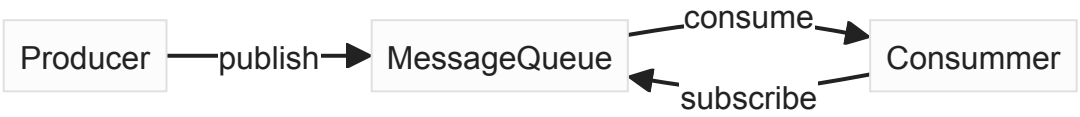
Data Centers

- Multiple centers
  - (Optional) How Netflix async multi-data centers replications
    - Data replication models
      - active-active
      - active-passive

- Cassandra for large-scale distributed data storage with eventual consistency
  - write
    - data replications across nodes - write to majority nodes is considered successful
    - Async propagation with data versioning
  - read
    - quorum of nodes agree on a value
    - data versioning
    - read repair - when multiple versions of data is detected, update all data to latest version

Message Queue

- Solve
  - further scale up system
    - de-couple producer/consumer components so each can scale up independently
  - async communication: decouple producers with consumers/backend processes, so producer don't need to wait for the result from consumers
  - Fault tolerant
- async / durable components
- serve as buffer to distribute async requests - flash sale/deal system (秒杀系统、排队买票系统)



- 书里Message Queue 的图有问题，应该是LB <--> Web Server (而不是LB直接连Queue)，然后Web Server可以Push request到Queue，Queue的Consummer可以Subscribe，并且异步处理Queue的message

Logging, metrics and automation

- MessageQueue for different logging / monitoring tools

Database Scaling

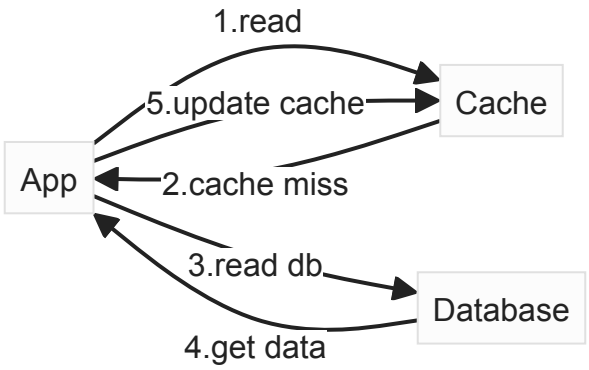
Different from data tier replicas, it's called Sharding

- each shard has same schema

- actual data on each shard is unique to the shard
- cautious to choose shard/partition key:
  - if can evenly distributed
- Challenges
  - Resharding
    - rapid data growth
    - uneven distribution - certain shard grows much faster than others
    - celebrity problem - each shard might requires further partition
  - joins and de-normalization
    - it's hard to join when with shard data, denormalize a little bit would solve
- Consider move more non-relational functions to NoSQL

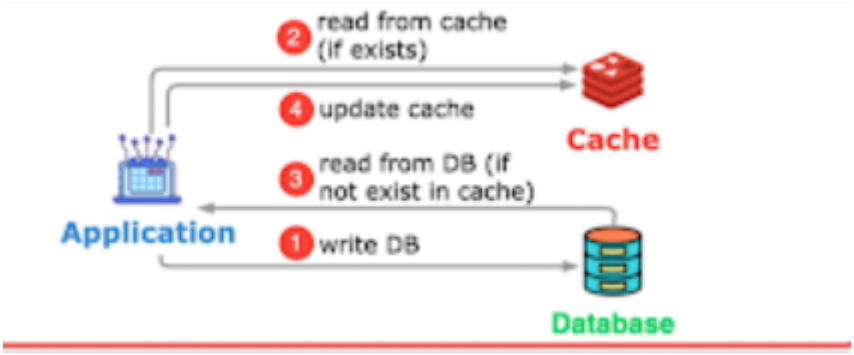
Caching Strategies

- Cache Aside
  - Pros
    - Update logic is on application level, easy to implement
    - cache only contains what the application requests for
  - Cons
    - Each cache miss results in 3 trips
    - data maybe stale if DB is updated directly

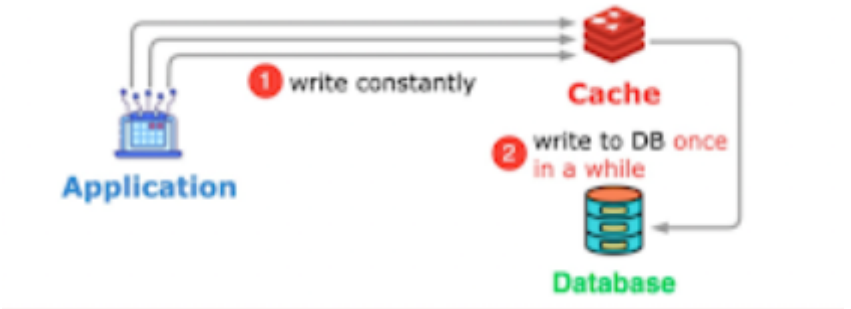


- Read Though
  - Pros:
    - application logic is simple
    - can easily scale the reads and only one query hits the DB
  - Cons
    - Data access logic is in cache, requiring writing a plugin

- Write Around
  - Pros
  - the DB is source of truth
  - lower read latency
  - Cons
  - Higher write latency - data is written to DB first
  - the data in cache maybe stale



- Write Back
  - Pros
  - lower write / read latency
  - cache and DB eventually consist
  - Cons
  - can be data loss if cache is down
  - infrequent data is also stored in cache



- Write Through
  - Pros
  - reads have lower latency
  - cache and db are in sync
  - Cons

- writes have higher latency - wait for DB writes to finish
- infrequent data is stored in cache

Questions

- columnar is a kind of NoSQL database, is ClickHouse / Postgres NoSQL?
  - **ClickHouse** is a columnar database, but it's not categorized as a NoSQL database.
  - **PostgreSQL** is a traditional SQL-based relational database and is not a NoSQL database.
- what is serialize/de-serialize data?
  - Serialization: convert data structure to a sequence of bits that can be stored in a file, memory buffer or transmitted across network, e.g.: JSON/XML
- When a master fail, how slave is prompted and how to handle data loss during promotion?
  - **Data Synchronization**: Before promotion, it’s crucial to ensure that the slave node is as up-to-date as possible with the master. However, there may be a window of data that was committed to the master but not yet replicated to the slave at the time of failure.
- How cache keep consistency - read paper Scaling Memcache at Facebook
- Is NoSQL like DynamoDB or Redis fast read / write both?
- single box vs. microservice (or distributed system), why Amazon get back ?
  - Link
  - ByteByteGo explain (Link)
- in master slave arch, how a server knows which database it goes for read? (Page 10)