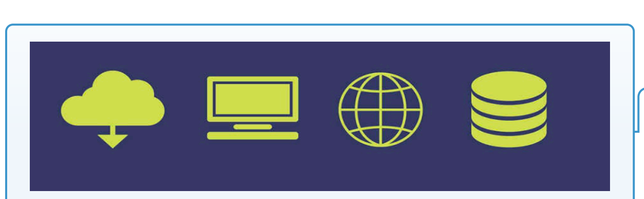
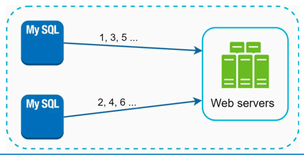


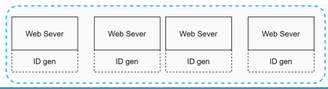
Alex Xu - System Design Interview (2nd ed.)

7. Design a Unique ID Generator in Distributed Systems

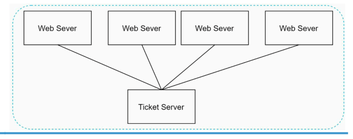
- Traditional single server DB with auto_increment is not large enough for distributed systems
- Step 1. Understand the problem and establish design scope
  - IDs must be unique
  - IDs are numerical values only
  - IDs fit into 64-bit
  - IDs are ordered by date
  - Ability to generate over 10,000 unique IDs per second
- Step 2. Propose high-level design and get buy-in
  - Possible options
    - Multi-master replication



      - Using DB's auto_increment feature but increase by k
        - where k is the number of database servers in use
      - Pros
        - It solves some scalability issues
      - Cons
        - Hard to scale with multiple data centers
        - IDs do not go up with time across multiple servers
        - It does not scale well when a server is added or removed
    - Universally unique identifier (UUID)
      - UUID is a 128-bit number used to identify information in computer systems
        - example: 09c93e62-50b4-468d-bf8a-c07e1040bfb2



      - ⚠️ *from Wikipedia: after generating 1 billion UUIDs every second for approximately 100 years would the probability of creating a single duplicate reach 50%*
      - Pros
        - Generating UUID is simple
          - No coordination between servers is needed
        - The system is easy to scale
          - each web server is responsible for generating IDs they consume
      - Cons
        - IDs are 128 bits long, but our requirement is 64 bits
        - IDs do not go up with time
        - IDs could be non-numeric
    - Ticket server
      - Flicker developed ticket servers to generate distributed primary keys



      - Pros
        - Numeric IDs
        - It is easy to implement, and it works for small to medium-scale ap-plications
      - Cons
        - Single point of failure
    - The only one option that fits out needs!
    - Twitter snowflake approach

      - Sign bit: 1 bit
        - It will always be 0. This is reserved for future uses
      - Timestamp: 41 bits
        - ⚠️ Milliseconds(!) since the epoch or custom epoch
        - We use Twitter snowflake default epoch 1288834974657, equivalent to Nov 04, 2010, 01:42:54 UTC
      - Datacenter ID: 5 bits
        - which gives us 2 ^ 5 = 32 datacenters
      - Machine ID: 5 bits
        - which gives us 2 ^ 5 = 32 machines per datacenter
      - Sequence number: 12 bits
        - For every ID generated on that machine/process, the sequence number is incremented by 1
        - ⚠️ The number is reset to 0 every millisecond
- Step 3. Design deep dive
  - Timestamp
    - The maximum timestamp that can be represented in 41 bits is 2 ^ 41 - 1 = 2199023255551 milliseconds (ms), which gives us: ~ 69 years
      - ⚠️ After 69 years, we will need a new epoch time or adopt other techniques to migrate IDs
  - Sequence number
    - Sequence number is 12 bits, which give us 2 ^ 12 = 4096 combinations
      - This field is 0 unless more than one ID is generated in a millisecond on the same server
      - In theory, a machine can support a maximum of 4096 new IDs per millisecond
        - 🟦 Q: So in order to be able to generate 10k per second we need at least 3 servers and load-balance our requests, right?
          A: 4096 per MILLI-second! so for 10k per second 1 server is more than enough!
- Step 4. Wrap up
  - few additional talking points to consider
    - ⚠️ Clock synchronization
      - Network Time Protocol is the most popular solution to this problem
    - ⚠️ Section length tuning
      - e.g. fewer sequence numbers but more timestamp bits
        - for low concurrency and long-term applications.
    - ⚠️ High availability
      - ID generator MUST be highly available!