Scale a system to millions of users

Satyajit Panda

19-May-2020

Assumptions

- This presentation is based on **first principles** of distributed scalable systems which can be appllied generally to any cloud ecosystem and data centers
- The numbers are for reference only and not absolute

Architectural tradeoffs

- Time: How long it takes you to setup
- Team: How productive your team will be with this decision
- Cost: How much you'll pay to AWS for these services
- Risk: How much down time / data loss / security risk you're exposed to
- Scale: How many users you can serve / how fast your app is

Approach

1. Design a system with small no. of users
2. **Benchmark/Load Test**
3. **Profile** for bottlenecks
4. Evaluate alternatives and trade-offs
5. Apply scalable design patterns, **repeat**

Define the Use case

- **User** makes a read or write request
  - **Service** does processing, stores user data, then returns the results
- **Service** needs to evolve from serving a small amount of users to millions of users
- **Service** has high availability

Constraints and assumptions

- Traffic is not evenly distributed
- Use of relational data
- Scale from one user to tens of millions of users
  - Users (1-10 users)
  - Users (10-100 users)
  - Users (100-1000 users)
  - Users (1000 to tens of thousands)
  - Users (Tens of thousands to one million)
  - Users (One million and beyond…)

Calculate usage

Users and read/writes

- 10 million users
- 1 billion writes per month
- 100 billion reads per month
- 100:1 read to write ratio
- 1 KB content per write

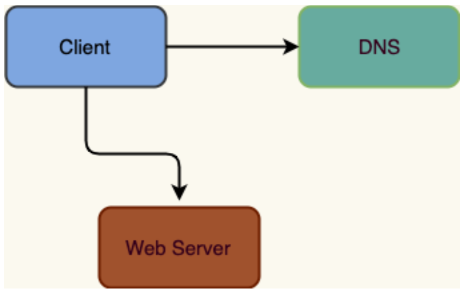
Back-of-the-envelope usage calculations

- 1 TB of new content/month
- 1 KB per write * 1 billion write/month
- 36 TB of new content in 3 years
- Assume most writes are create instead of updates
- 400 writes per second on average
- 400,00 reads per second on average

Handy conversion guide

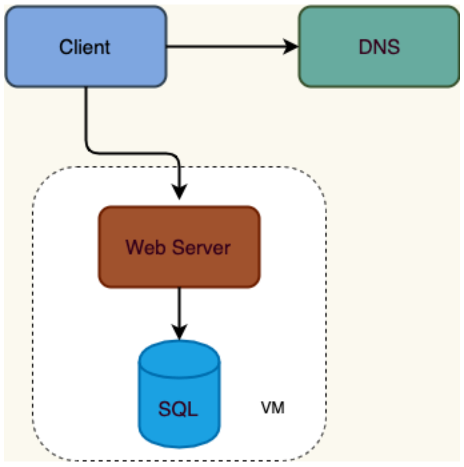
- 2.5 million seconds/month
- 1 request/second = 2.5 million requests/month
- 40 requests/second = 100 million requests/month
- 400 requests/second = 1 billion requests/month

Create a high level design



Design core components for 1-10 users

Design core components for 1-10 users



Use case: User makes a read or write request

Goals

- With only 1-10 users, you only need a basic setup
  - Single box for simplicity
  - Vertical scaling when needed
  - Monitor to determine bottlenecks

Start with a single box

- **Web server** on EC2
  - Storage for user data : **MySQL Database**
- Start with SQL, consider NoSQL
- Assign a public static IP through elastic IP
- Add Route 53 DNS to map the domain name to public IP
- Secure the web server
  - Open up only necessary ports like 80(HTTP),443(HTTPS),22(SSH)
  - Prevent the web server from initiating outbound connections

---

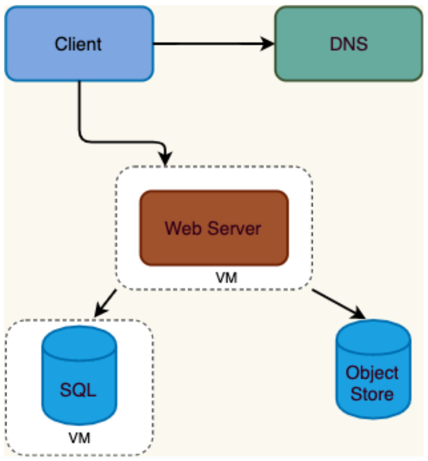
- Use **Vertical Scaling**:
  - Simply choose a bigger box
  - Use basic monitoring to capture metrics and determine bottlenecks: CPU, memory, IO, network, etc
  - CloudWatch, top, nagios, statsd, graphite, etc
- **Tradeoffs**:
  - Scaling vertically keeps on getting expensive
  - No redundancy/failover

AWS EC2 instance Cost Spectrum

	Cheapest	Most expensive
Name	t3.nano	x1e.32xlarge
Specs	2 vCPUs/0.5 GiB	128 vCPUs/3,904 GiB
Cost	$3.796/month	$19215/month

Scale the design

Users(10-few hundreds)



Assumptions

- User count picking up and the load is increasing
- MySQL taking most of the memory and CPU resources
- The user content is filling up disk space
- Vertical scaling is geting expensive and doesn't allow for independent scaling of the **MySQL Database** and **Web Server**.

Goals

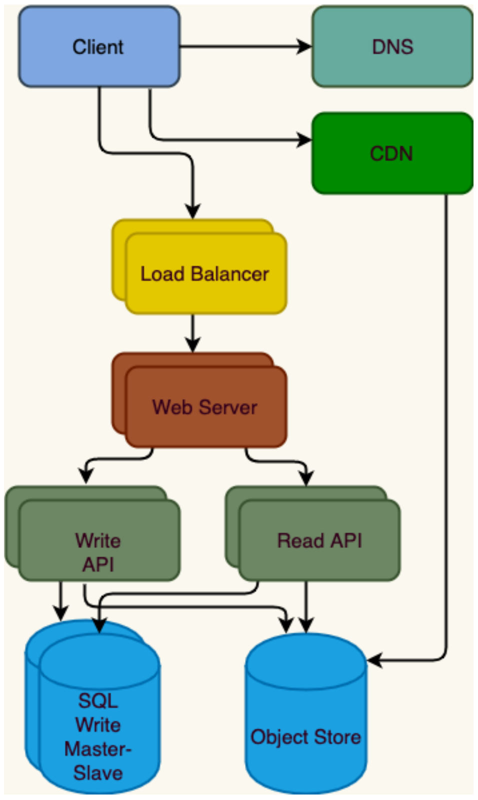
- Lighten load on the single box and allow for independent scaling
  - Introduce **Object Store**
  - Move the **MySQL Database** to a separate box
- **Disadvantages**

  - Added complexity
  - Additional security measures
  - AWS cost tradeoff

- Store static content separately
- Consider using a managed **Object Store** like S3 to store static content
  - Highly scalable and reliable
  - Server side encryption
- Consider using a service like RDS to manage the **MySQL Database**
  - Simple to administer, scale
  - Multiple availability zones
  - Encryption at rest

- **Secure the system**
  - Encrypt data in transit and at rest
  - Use a Virtual Private Cloud
    - Create a public subnet for the single Web Server so it can send and receive traffic from the internet
    - Create a private subnet for everything else, preventing outside access
    - Only open ports from whitelisted IPs for each component

Users (Few hundreds to few thousands)



Assumptions

- Single **Web Server** is a bottleneck now during peak hours, resulting in
  - Slow responses
  - Downtime
- Introduce higher availability and redundancy

Goals

- Use **Horizontal Scaling** to handle increasing loads and to address single points of failure
  - Add a Load Balancer such as AWS ELB or HAProxy
    - Amazon ELB is highly available
    - For HAProxy, set up multiple servers in active-active or active-passive in multiple availability zones for improved availability
    - Terminate SSL on the Load Balancer to reduce computational load on backend servers and to simplify certificate administration

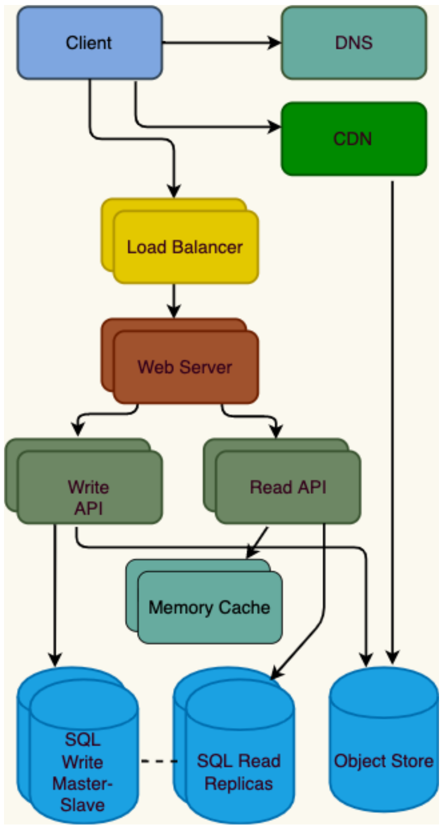
---

- Use multiple Web Servers spread out over multiple availability zones
- Use multiple MySQL instances in Master-Slave Failover mode across multiple availability zones to improve redundancy

---

- Separate out the **Web Servers** from the **Application Servers**
  - Scale and configure both layers independently
  - **Web Servers** can run as a **Reverse Proxy**
  - Add a group of **Application Servers** handling **Read APIs** and let another group handle **Write APIs**
- Move static (and some dynamic) content to a **Content Delivery Network (CDN)**such as CloudFront to reduce load and latency

Users (One thousand to tens of thousands)



Assumptions

- Our Benchmarks/Load Tests and Profiling show that we are read-heavy (100:1 with writes) and our database is suffering from poor performance from the high read requests.

Goals

- The following goals attempt to address the scaling issues with the MySQL Database
  - Based on the Benchmarks/Load Tests and Profiling, you might only need to implement one or two of these techniques
- Move the following data to a Memory Cache such as Elasticache to reduce load and latency:
  - Frequently accessed content from MySQL
    - First, try to configure the MySQL Database cache before implementing a Memory Cache

---

- Session data from the Web Servers
  - The Web Servers become stateless, allowing for Autoscaling
- Reading 1 MB sequentially from memory takes about 250 microseconds, while reading from SSD takes 4x and from disk takes 80x longer

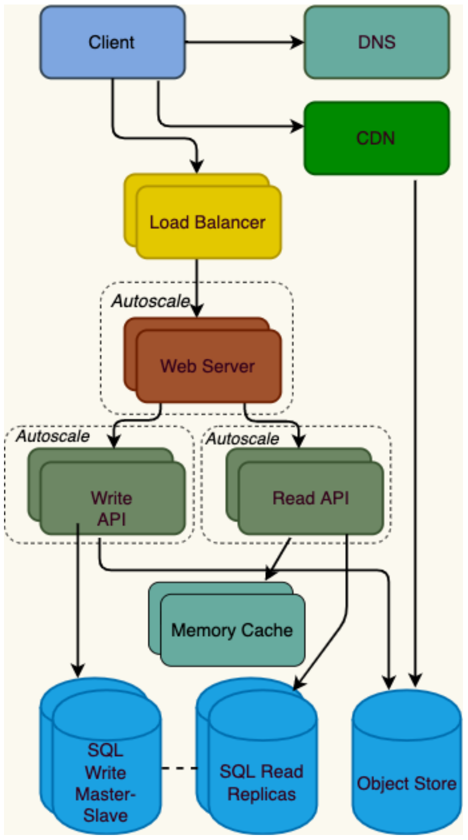
- Add MySQL Read Replicas to reduce load on the write master
- Add more Web Servers and Application Servers to improve responsiveness

---

Add MySQL read replicas

- In addition to adding and scaling a Memory Cache, MySQL Read Replicas can also help relieve load on the MySQL Write Master
- Add logic to Web Server to separate out writes and reads
- Add Load Balancers in front of MySQL Read Replicas (not pictured to reduce clutter)
- Most services are read-heavy vs write-heavy

Users (Tens of thousands to one million)



Assumptions

- Our Benchmarks/Load Tests and Profiling show that our traffic spikes during regular business hours and drop significantly when users leave the office. We think we can cut costs by automatically spinning up and down servers based on actual load. We're a start up so we'd like to automate as much of the DevOps as possible for Autoscaling and for the general operations.

Goals

- Add Autoscaling to provision capacity as needed
- Keep up with traffic spikes
- Reduce costs by powering down unused instances

- Automate DevOps
  - Chef, Puppet, Ansible, etc
  - Continue monitoring metrics to address bottlenecks
  - Host level - Review a single EC2 instance
  - Aggregate level - Review load balancer stats
  - Log analysis - CloudWatch, CloudTrail, Loggly, Splunk, Sumo
  - External site performance - Pingdom or New Relic
  - Handle notifications and incidents - PagerDuty
  - Error Reporting - Sentry

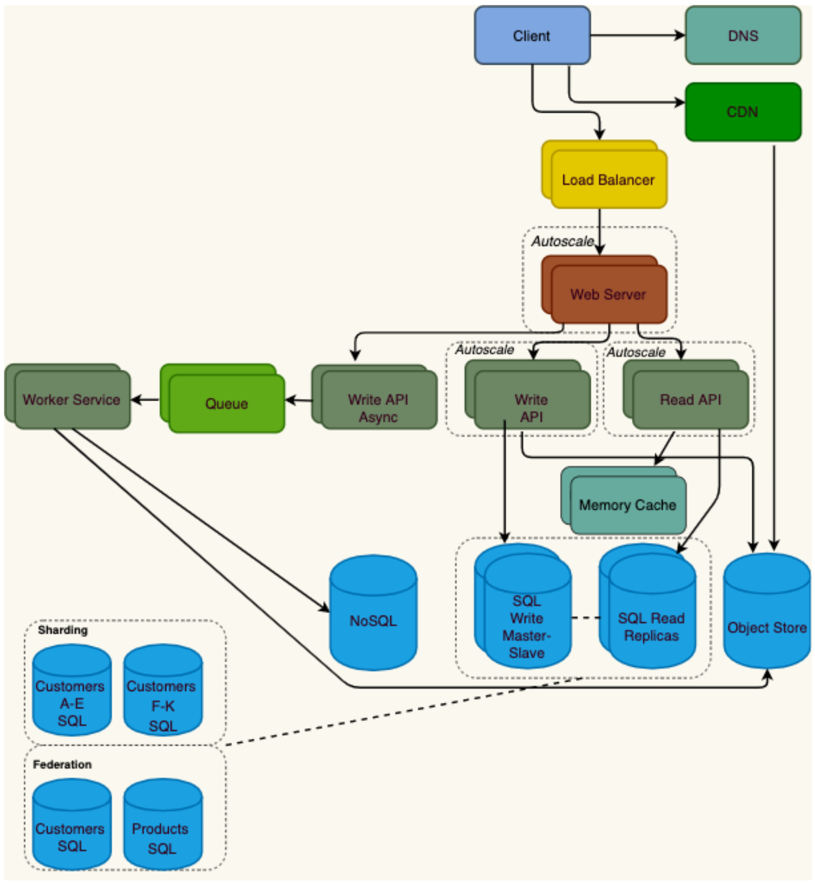
Add autoscaling

- Consider a managed service such as AWS Autoscaling
  - Create one group for each Web Server and one for each Application Server type, place each group in multiple availability zones
  - Set a min and max number of instances
  - Trigger to scale up and down through CloudWatch
    - Simple time of day metric for predictable loads
    - Metrics over a time period:CPU load | Latency | Network traffic | Custom metric

---

- Disadvantages
  - Autoscaling can introduce complexity
  - It could take some time before a system appropriately scales up to meet increased demand, or to scale down when demand drops

Users (One million and beyond…)



Assumptions

- As the service continues to grow towards the figures outlined in the constraints, we iteratively run Benchmarks/Load Tests and Profiling to uncover and address new bottlenecks.

Goals

- We'll continue to address scaling issues due to the problem's constraints:
- If our MySQL Database starts to grow too large, we might consider only storing a limited time period of data in the database, while storing the rest in a data warehouse such as Redshift
  - A data warehouse such as Redshift can comfortably handle the constraint of 1 TB of new content per month

---

- With 40,000 average read requests per second, read traffic for popular content can be addressed by scaling the Memory Cache, which is also useful for handling the unevenly distributed traffic and traffic spikes
  - The SQL Read Replicas might have trouble handling the cache misses, we'll probably need to employ additional SQL scaling patterns
- 400 average writes per second (with presumably significantly higher peaks) might be tough for a single SQL Write Master-Slave, also pointing to a need for additional scaling techniques

---

- SQL scaling patterns include:
  - Federation
  - Sharding
- Denormalization
- SQL Tuning
- To further address the high read and write requests, we should also consider moving appropriate data to a NoSQL Database such as DynamoDB.

- We can further separate out our Application Servers to allow for independent scaling. Batch processes or computations that do not need to be done in real-time can be done Asynchronously with Queues and Workers:

- For example, in a photo service, the photo upload and the thumbnail creation can be separated:
  - Client uploads photo
  - Application Server puts a job in a Queue such as SQS
  - The Worker Service on EC2 or Lambda pulls work off the Queue then:
    - Creates a thumbnail
    - Updates a Database
    - Stores the thumbnail in the Object Store