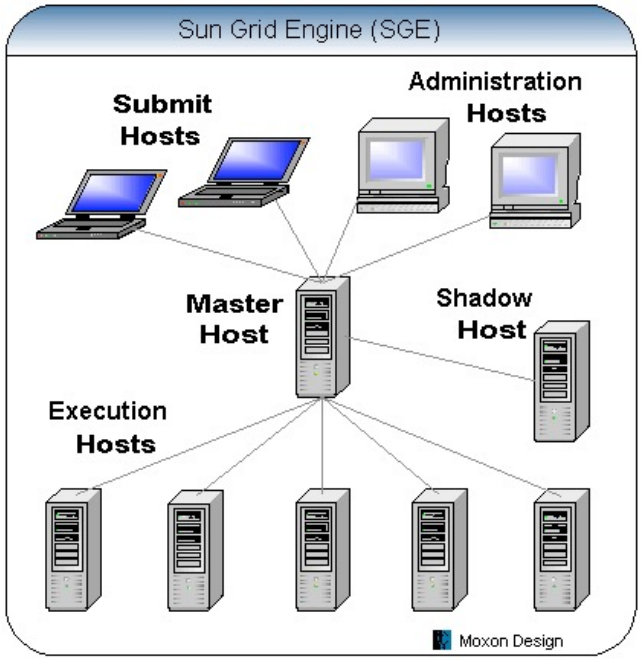
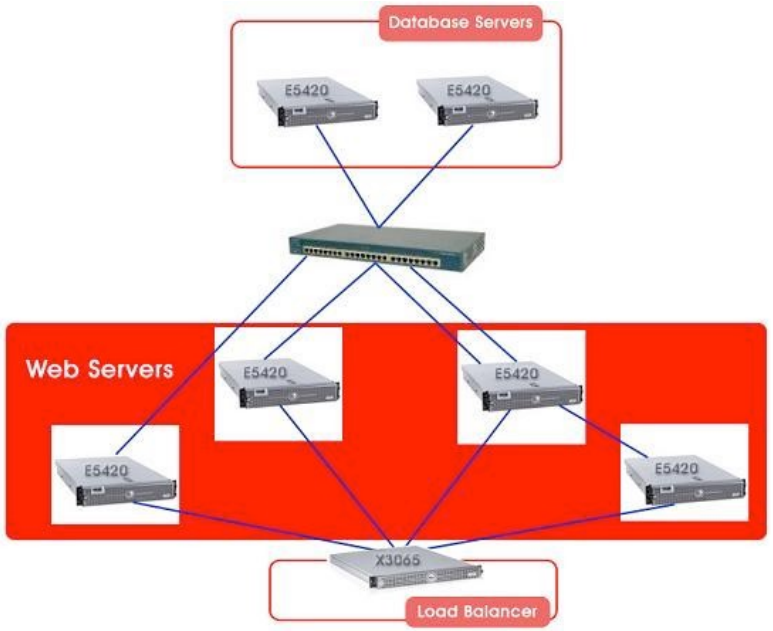
Cluster Computing

Architectures

Job Parallelism



Request Parallelism



Outline

- Replication
- Load balancing
- Self provisioning / auto scaling
- Replica control
- Sharding
- Caching

Replication

- **Fault tolerance**
  - High availability despite failures
  - If one replica fails, there is another one that can take over
- **Throughput**
  - Support a large user base
  - Each replica handles a fraction of the users
  - Scalability
- **Response time**
  - Support a geographically distributed user base
  - There is a replica close to the user
  - Fast local access

Stateless vs. Stateful

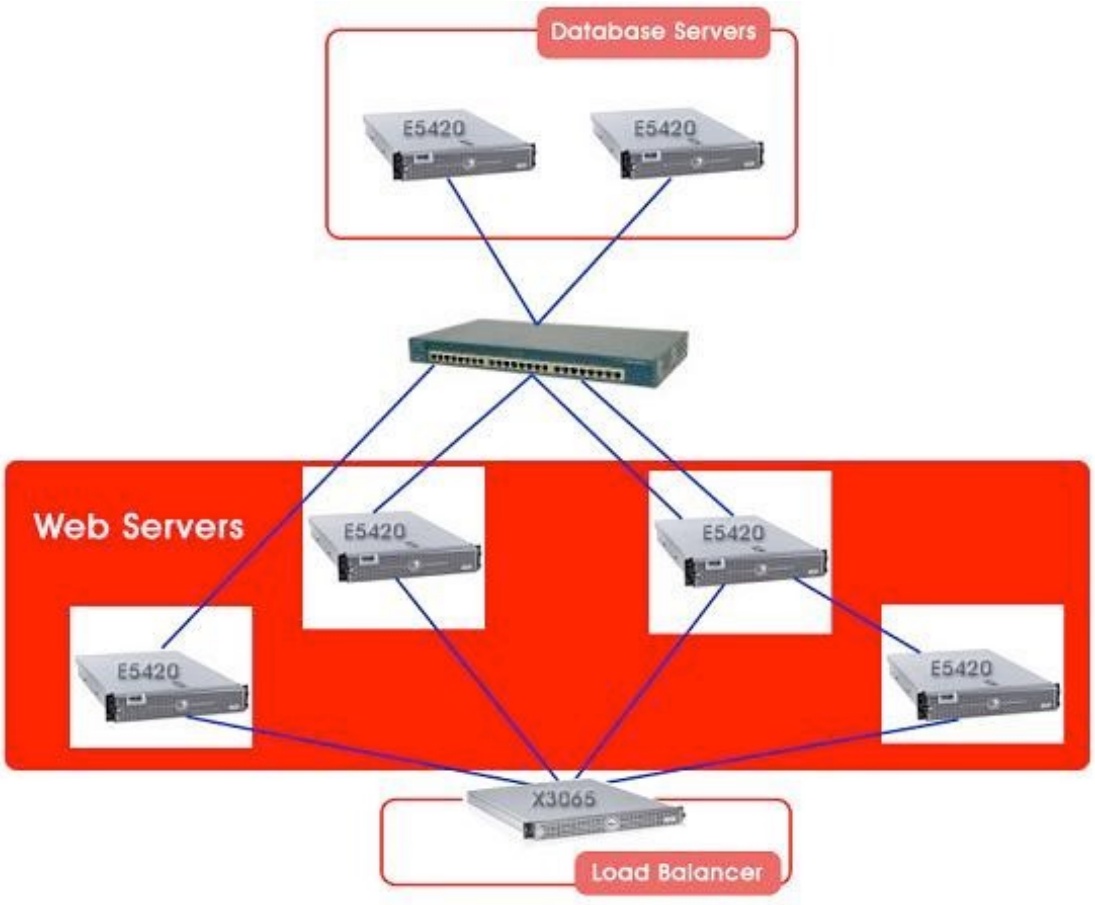
- **Stateless**
  - Services that require mainly computing
  - State can be regenerated on demand
  - Easy to replicate
  - Example: web servers

- **Stateful**
  - Services uses business critical data
  - Good performance depends on state availability
  - Harder to replicate
  - Example: database servers

Replication Example

Stateful

Stateless



Issues

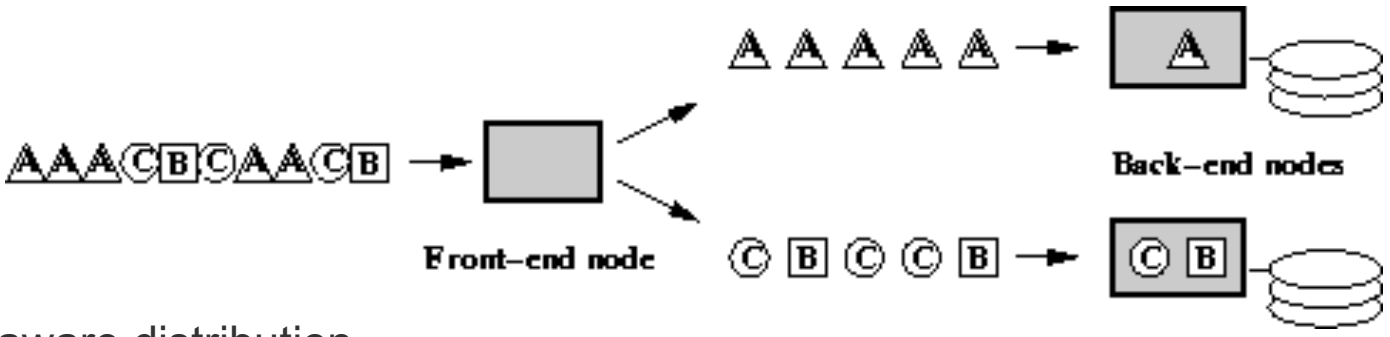
- Load balancing
- Provisioning
- State
- Fault tolerance

Load Balancing

- **Determines where requests will be executed**
- **Blind techniques**
  - Require minimum or no state on load balancer
  - Strategies:
    - Random
    - Round robin
- **Load aware**
  - Consider load on replicas when routing requests
  - Different request may generate different load
  - Replica nodes may have different capacity
  - Strategies:
    - Shortest queue first
      - Simple to implement, just track number of active requests
    - Least loaded
      - Replicas need to periodically provide load info to load balancer

Load Balancing

- **Application Aware**
  - Uses knowledge of the application
  - Strategies:
    - Shortest execution length
      - Profile request types to generate execution time estimates
      - Load estimates load at replica by keeping tracks of pending requests and their type, i.e., determines application-specific load metric.
      - Routes to less loaded replica
    - Locality-aware request distribution (LARD)
      - Takes into account data accessed by requests, and routes similar request to the same replica to increase cache hit rates



    - Conflict-aware distribution
      - Execute potentially conflicting requests on same replica for early conflict detection

Load Balancing Comparison

- **Blind**
  - General
  - Easy to implement
  - Suboptimal
- **Load aware**
  - Better performance
  - Require more state
- **Application aware**
  - Best performance
  - Require even more state
  - Harder to implement
  - Brittle – need to change if application changes

AWS Elastic Load Balancer

- **Balances load between EC2 instances**
- **Can distribute load across availability zones**
- **Within an availability zone:**
  - Round-robin among the least-busy instances
  - Tracks active connections per instance
  - Does not monitor CPU
- **Supports the ability to stick user sessions to specific EC2 instances.**

Self Provisioning

- **Automatically adjust the worker pool or number of replicas to the current workload**
  - Add workers/replicas when load increases
  - Retire replica/workers when under load

- **Approaches**
  - Reactive
    - Reconfigure when load threshold reached
  - Proactive
    - User prediction mechanism to trigger reconfiguration
    - Reconfigure before saturation
    - Based on time series or machine learning approaches

Self Provisioning

- **Retiring replicas/workers**
  - Move instance load to rest of system
    - All new request go to other nodes
  - Wait for outstanding transactions to commit
  - When idle, release

- **Adding replicas/workers**
  - Boot new node
  - Transfer state/database to new replica (optional)
  - Add replica to pool

Self Provisioning Considerations

- **Load metrics**
  - Use low level system metrics to determine node utilization
    - CPU, I/O utilization
  - Use application level metrics
    - Response time, e.g., transaction completes within X milliseconds.

- **Cost/latency prediction**
  - How long and how expensive it is to add a replica/worker
  - Reactive: the higher the latency the lower the configuration threshold
  - Proactive: the higher the latency the farther into the future we need to predict

- **Monitoring**
  - Monitoring isn’t free
  - The more data that we collect, the higher the impact on the system

AWS Auto Scaling

- Scale automatically according to user-defined conditions
- Enabled by CloudWatch
- Create scale up/down policy
- Associate scaling policy with CloudWatch alarm

Replica Control

- **Task of keeping data copies consistent as items are updated**
- **There is a set of database nodes R^A, R^B, …**
- **Database consist of set of logical data items x, y, ….**
- **Each logical item x has physical copies x^A, x^B,….**
  - Where x^A resides in R^A
- **A transaction is a sequence of read and write operation on logical data items**
- **The replica control mechanism maps the operation on the logical data items onto the physical copies**

Read-One-Write-All (ROWA)

- **Common replica control method**
- **Read can be sent to any replicas**
  - Logical read operation $r_i(x)$ on transaction T_i
  - Mapped to $r_i(x^A)$ on one particular copy of x
- **Updates performed on all replicas**
  - Logical write operation $w_i(x)$ on transaction T_i
  - Mapped to $w_i(x^A)$, $w_i(x^B)$, … on one particular copies of x

Read-One-Write-All (ROWA)

- **Common replica control method**
- **Read can be sent to any replicas**
  - Logical read operation $r_i(x)$ on transaction T_i
  - Mapped to $r_i(x^A)$ on one particular copy of x
- **Updates performed on all replicas**
  - Logical write operation $w_i(x)$ on transaction T_i
  - Mapped to $w_i(x^A)$, $w_i(x^B)$, … on one particular copies of x
- **ROWA works well because on most applications reads >> writes**

Primary Copy vs. Update Anywhere

- **Primary copy**
  - All updates are executed first on a single node, the primary
  - Advantages: simpler to implement
  - Disadvantages: primary can be come bottleneck
- **Update Anywhere/Everywhere**
  - Updates and read only request can be sent to any replica
  - Advantages: potentially more scalable
  - Disadvantages: harder to guarantee consistency

Processing Write Operations

- **Writes have to the executed on all replicas**
- **Write processing is the main overhead of replication**
- **Symmetric update processing**
  - SQL statement is sent to all replicas
  - All replicas parse the statement, determine the tuples affected, and perform the modification/deletion/insertion.
  - Pros: Reuse existing mechanisms
  - Cons: Redundancy
    Execution has to be deterministic. Consider and update that sets a timestamp
- **Asymmetric update processing**
  - SQL statement is executed locally on a single replica
  - Writeset (extracted changes) sent to other replicas
    - Identifier and after-image of each updated record
  - Pros: Efficiency
  - Cons: Harder to implement

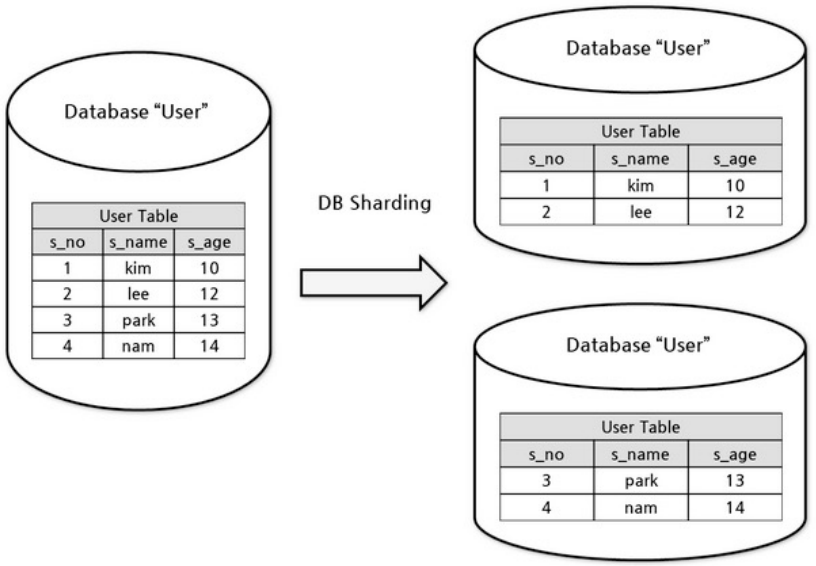
MySQL Replication

- ROWA
- Primary copy
- Eager and lazy replication
- Symmetric and asymmetric update processing
- Full and partial replication

Sharding

- **Challenges:**
  - Very large working set
    - Slows reads
    - Facebook/Google user table
  - Too many writes

- **Solution:**
  - Partition the data into ***shards***
  - Assign shards to different machines
  - Denormalize the data



Sharding Strategies

- **Range-based partitioning**
  - E.g., username from a to m assigned to shard 1, n to z to shard 2
  - Pros: simple
  - Cons: hard to get load balancing right

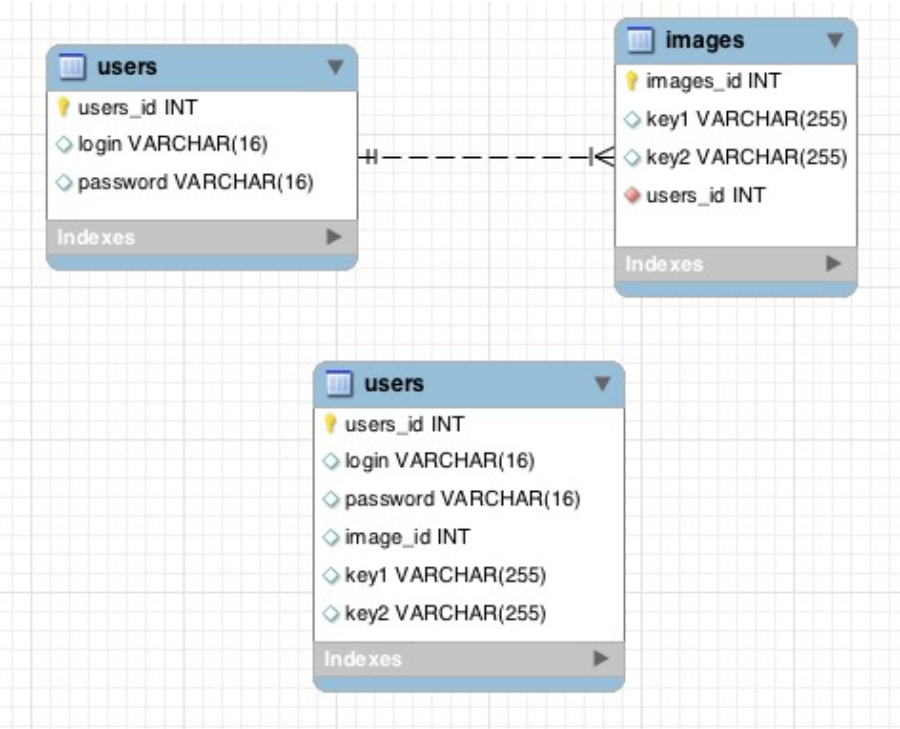
- **Directory-based partitioning**
  - Lookup service keeps track of partitioning scheme
  - Pros: Flexibility
  - Cons: Lookup service may become bottleneck

- **Hash-based partitioning**
  - Compute hash of key.
  - Different shards responsible for different hash ranges
  - Pros: Good load balancing
  - Cons: A little more complex,

Hash-based Partitioning

Denormalization

- **Data that is access together is stored together**
  - E.g., multi valued properties in AppEngine datastore entities
- **Eliminates costly joins**
- **May require replication**
  - E.g., a comment may be stored on the commenter's and commentee's profile




```
select * from users, images
where users.user_id = images.user_id and
      users.user_id = 10;
```

```
select * from users where user_id = 10;
```

Sharding

- **Pros**
  - High availability
  - Faster queries
  - More write bandwidth

- **Cons**
  - Queries get more complicated
    - Need to figure out which shard to target
    - May need to join data from multiple shards
  - Referential integrity
    - Foreign key constrains are hard to enforce
    - Not supported in many databases

NoSQL Examples

- MongoDB
- Dynamo
- Google Datastore
- Non-relational data model
- Limited transactional support
- Sharding

Caching

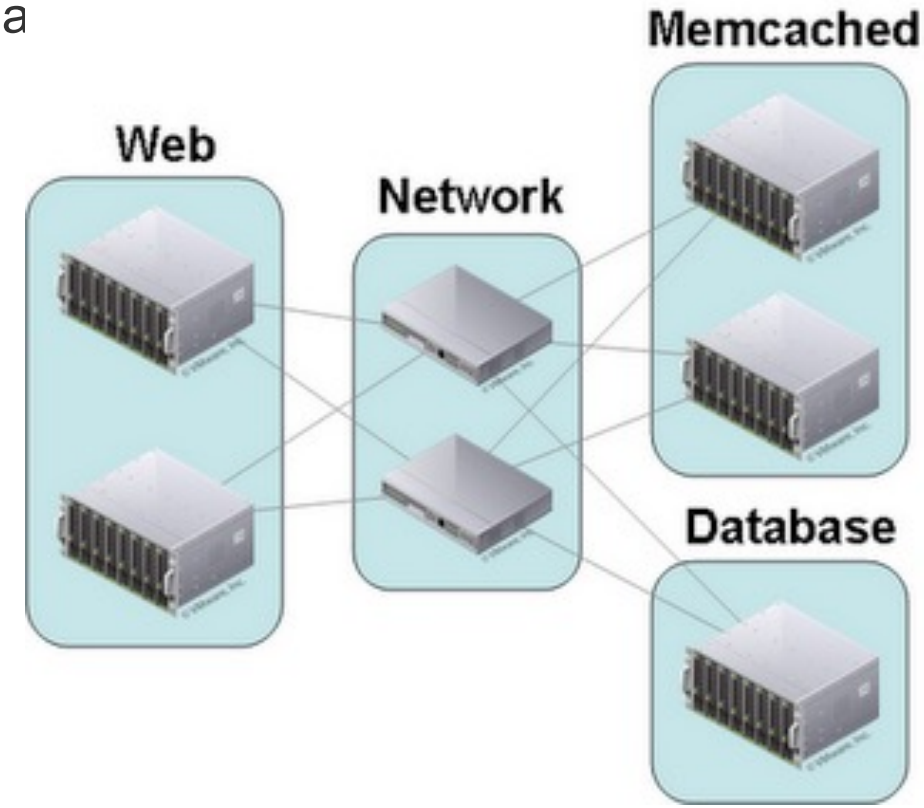
- **Objective:**
  - Reduce load on storage server
  - Reduce latency to access data
- **Store recent/hot/frequent data in RAM**
- **Volatile**
- **Fast read/write access**

memcached

- **A distributed memory cache implementation**
- **Provides a key-value store interface**

  put (key,value)

  value = get(key)
- **Scalable and consistent temporary storage**
  - Secondary system that provides fast a
  - Data stored reliably somewhere else



Usage Model

- **Data accesses**

  get value from memcached

  if cache miss

    fectch data from datastore

    put value on memcached

  operate on value

- **Data updates**
  - Possible to overwrite value with new data
  - No transactional way to do this
  - Update may succeed in datastore and fail in memcache
  - Updates may happen at different order in datastore and memcache
  - Instead: Invalidate item on update
    Fetch fresh value on next access