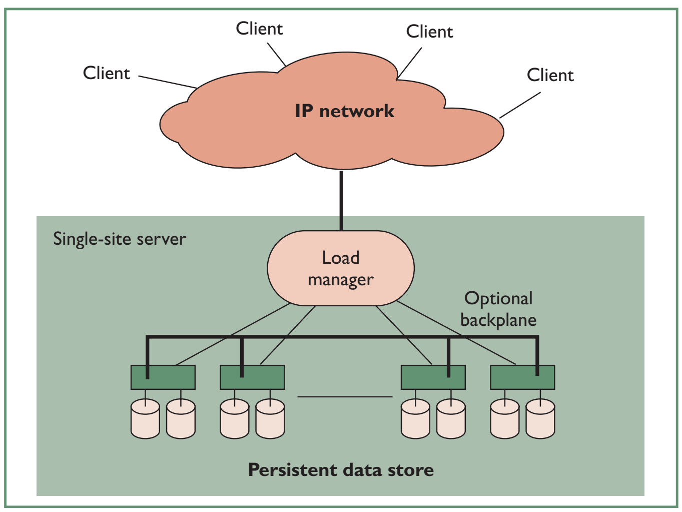
Scaling services

1. Giant scale services
   a. Questions from reviews
      i. Uptime vs recovery time?
      ii. Unhelpful focus on read-only services
      iii. Are systems more network-bound than disk bound?
         1. It is when you introduce caching
      iv. How does harvest relate to non-search systems?
         1. Reduce amount of data for adds, recommendations
         2. Precision of # of messages in mailbox
         3.
   b. Background:
      i. Eric Brewer and some grad students founded inktomi as a search engine using a google-style architecture: commodity workstations and networks (myrinet cluster)
      ii. We read his papers because he writes about his experiences (few others do) and writes for our community
   c. What problems addressed in this paper?
      i. Basic architecture
         1. Load-balancing front end, back-end nodes:, separate data store



         2. Best-effort service
         3. Where not appropriate?
            a. E-commerce: want to store orders, credit card transactions
         4. Why clusters?
            a. Only way to scale to the whole planet
            b. Cheap to buy
            c. Incrementally scalable
            d. Independent failures of small components
         5. Cluster architecture:
            a. Use "symmetric design" – really means homogeneous
      ii. Load management: LARD & consistent hashing type approaches

iii. High availability

d. Availability
   i. Metrics
      1. MTTF/MTBF = time between failures
      2. MTTR = time to repair
         a. Restart app after app crash
         b. Reboot after system crash
         c. Repair /replace hardware after hardware crash
         d. Move workload to another machine
         e. QUESTION: Which should you try to improve – MTTR or MTTF?
            i. Depends on how long computations run for – if short, then little is lost from a failure
      3. Availability/uptime = (MTBF-MTTR)/MTBF = fraction of time you are available to serve data
         a. In a setting with multiple data centers and independent failures, what does this mean?
            i. What a single user sees?
               1. If the internet goes down on their side, they see zero
            ii. Aggregate: of all requsts/ what fraction served?
      4. Yield = # queries completed / # queries offered
         a. Aggregate availability
         b. QUESTION: How define for google docs or gmail?
      5. Harvest = data available (how much data used for query) / complete data
         a. Q: how use in email?
            i. What fraction of inbox/total messages available?
         b. Q: how use in ecommerce?
            i. Reduce number of suggestions
         c. Q: how use in ebay?
            i. Simplified rendering of pages, fewer suggestions or data per page
         d. Q: how use in new york times online?
            i. Simplified pages, less dynamic content

e. Architectures for availability:
   i. Replication: store multiple copies of data
      1. Q: what happens on failure?
         a. Yield goes down – fewer servers to answer results
         b. Harvest stays same (all data still available)
   ii. Partition: split data into smaller chunks
      1. Q: what happens on failure?
         a. Harvest goes down – cannot see all data

        b. Yield stays same (copies of other data stay same)
    iii. QUESTION: What does consistent hashing /LARD do?
        1. Mostly partitioning, replication only for super-hot data
    iv. NOTE: everybody does both
    v. Replication and read/write data
        1. For read-only data, replication adds scalability – can serve more than possible on a single machine
        2. For read/write data, write throughput limited to what a single machine can handle
            a. Must write to all machines, so replication does not improve throughput
            b. Must partition to the point where load can be handled by a single machine

f. Scalability
    i. DQ principle
        1. Data per query X queries per second = constant for a given cluster/architecture
            a. This is the amount of data you need to process per second, driven by number of machines, disk throughput, network throughput, memory capacity (for caching)
        2. DQ of a cluster is a capacity metric
            a. DQ of a workload is the demand on the cluster. You hope the DQ of the cluster is higher than the DQ of the demand
    ii. How do replication/partitioning and failures affect DQ?
        1. Replication: increase # of queries per second by having more machines answer each query
            a. Failure leads to fewer queries per second
        2. Partitioning: increase amount of data by having more machines store data
            a. Failure leads to less data per query
        3. Result: a failure in either case reduces aggregate capacity the same way
        4.

Table 1. Overload due to failures.

Failures	Lost capacity	Redirected load	Overload factor
1	$\frac{1}{n}$	$\frac{1}{n-1}$	$\frac{n}{n-1}$
k	$\frac{k}{n}$	$\frac{k}{n-k}$	$\frac{n}{n-k}$

        5. What happens to the load? Must send it somewhere else (with replication)
            a. If lose 1/n machines, then each other machine must add 1/(n-1) more capacity (with replication)

i. 5 machines, 1 crashes -> each machine has ¼ more capacity (divide 1 machine over 4)

b. Other machines have n/(n-1) load (5/4 in our example)

g. What happens at overload?
    i. Overload can happen when unexpected failures (data center) or unexpected workloads (Slashdot effect)
    ii. What bad thing happens?
        1. Congestion collapse: latencies get so long everybody times out and retries
    iii. How can you handle?
        1. Must reduce DQ of the load
            a. Queries per second: admission control
                i. Fail low-priority queries
            b. Data per query: incomplete answers
                i. Fewer email messages displayed (in email)
                ii. Fewer tail search results
                iii. Fail complex queries early (lower average data per query)
                iv. Stale data (more caching)
h. Online evolution
    i. Cannot take down an internet service (although AOL used to go down for a few hours every week
    ii. Key question: can versions co-exist?
    iii. Solutions:
        1. Fast reboot: reboot all machines at the same time during off peak hours
            a. Avoid incompatibilities
        2. Rolling upgrade: upgrade in waves, take down 1/#waves at a time
            a. Longer latency, lower impact
            b. Need to support co-existence of versions
        3. Big flip
            a. Do half the machines at a time, switch from old to new with network switch
    iv. Must support lowered throughput during upgrade, or do during off-peak hours
i. Why read
    i. See how load balancing fits into picture
    ii. See how make service infinitely scalable
        1. Replicate, partition
        2. Plan for added load after failure
    iii. See fault tolerance techniques
        1. MTTR vs MTTF
    iv. See issues

            1. Upgrades
            2. Capacity (throughput) = DQ

2. Dynamo
    a. Questions from reviews?
        i. Gossip-based protocol
            1. Does it limit size? They say they have a size limit elsewhere
    b. Why read this paper?
        i. Introduction to a ton of ideas
            1. merkle trees
            2. quorum protocols
            3. gossip protocols
            4. vector clocks
            5. Anti-entropy replication
            6. CAP theorem
    c. Looks at issues of partitioning & replication & fault tolerance & load specifically
    d. What are key ideas
        i. Define the appropriate service
            1. key-value store vs RDBMS
        ii. Define the appropriate consistency metric
            1. Generally, what is the loosest thing your application can handle?
                a. Dynamo:
                    i. No lost data or silent overwrites
                    ii. Always writeable
        iii. Partition your data
            1. Hash on the key of an object
            2. Assign servers to hash buckets explicitly (consistent hashing)
            3. Virtual servers to spread load more evenly
        iv. Replicate your data
            1. Write data to some number of nodes
            2. Read from some number of nodes
            3. If you can guarantee they overlap, then you have consistency
            4. Assign a coordinator among top N replicas
                a. helps with consistency because it knows of previous versions of data
        v. Handle failures
            1. Send reads/writes somewhere else
                a. hinted handoff
            2. Propagate changes back on recovery
                a. merkle trees & anti-entropy for detecting missing changes
        vi. Keep track of members
            1. Explicit add/remove of nodes by admins
                a. permanently changes the home of data
            2. Failure detector & periodic retry for temporary outages
                a. Temporarily sends reads/writes to next nodes down ring

vii. Locate data
    1. Load balancer to ring member for dumb clients
        a. adds layer of indirection but removes complexity of client
    2. Smart clients know which servers to contact
        a. reduces latency at a complexity cost

e. Big idea:
    i. Build the simplest useful system
        1. Reduce the guarantees to the ones you cannot provide at a higher level
            a. write availability
        2. Push complexity out of the service to client when feasible
            a. Managing conflicts
        3. Leverage centralization when possible
            a. assignment of tokens to servers
            b. Seeds