System Support and Design Issues in Online Query Processing

•Tao Yang 293S, 2022

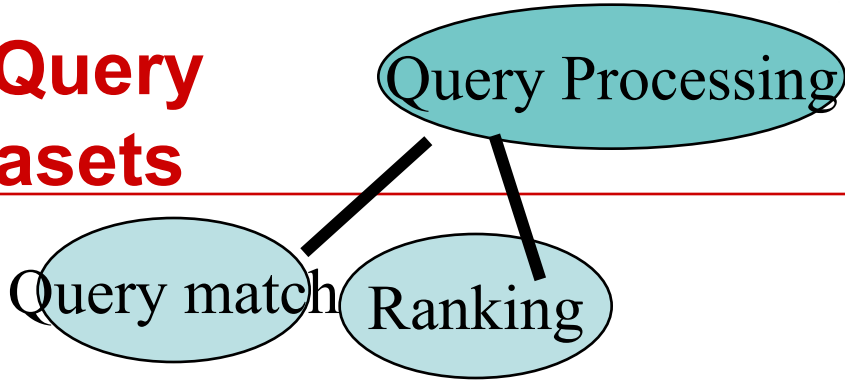
Content

- System support and design tradeoffs in online query processing
  - Objective: fast response, high throughput, and high availability
- Example of online architecture
- Building/running services on the cloud
  - Elastic computing. Software as services

Online Data for Search: Inverted Index and Auxiliary Structures

- **Inverted lists usually stored together in a single file for efficiency**
  - Term statistics stored at start of inverted lists
- ***Vocabulary* or *lexicon***
  - Contains a lookup table from index terms to the byte offset of the inverted list in the inverted file
  - Either hash table in memory, key-value stores, or B-tree for larger vocabularies
- **Document-oriented information**
  - E.g. Document quality score, freshness indicator, page text content
  - In-memory hashtable, key-value stores
- **Other information**
  - Collection statistics. Web host information

Design Consideration of Query Processing for Large Datasets



- Go through all postings of queries words
- Conduct matching & ranking

- **Estimate I/O cost**
- **Memory cache for storing frequently accessed items**
  - Cache size requirement: Is there enough memory?
  - Does program exhibit cache locality?
- **Distribute data to multiple machines for parallel processing**
  - Distribute disk data to p machines evenly
  - Distributed memory data to p machines evenly

System Challenges for Online Services

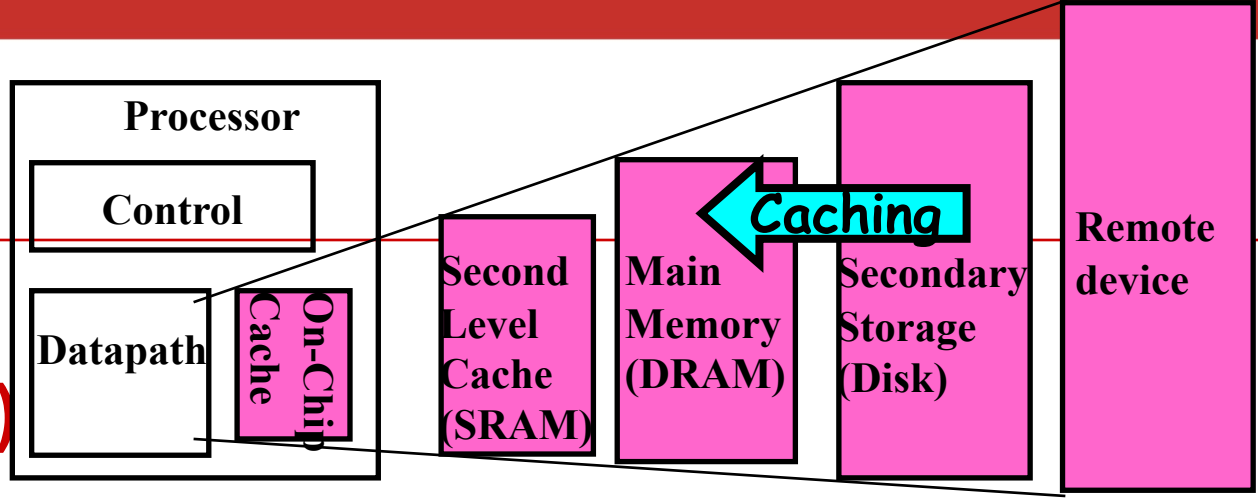
- **Challenges/requirements for online services:**
  - Each system needs tens or hundreds of subservices, running on hundreds or thousands of machines if not more.
  - Low response time, high throughputs
  - Data intensive, need to consider impact of cache, memory, disk I/O
  - Huge amount of data, requiring
    - Large-scale clusters.
    - Incremental scalability.
  - 7×24 availability with fault tolerance:
    - Operation errors, Software bugs, Hardware failures
  - Resource management, QoS for load spikes.
- Careful **design planning** in architecture and system support choices for reliable/scalable online services

Response Time vs Concurrency for Search Query Processing

- Backend response time requirement: ~200 ms per request
- Throughput requirement: number of requests second per machine
  - 100 Requests/second per machines
  - → 10 machines 1000 requests →86 million requests/day
- Rules of thumb
  - Writes are expensive. Reads are cheap (Search engine does read most of time)
  - Access HDD is expensive and a few are allowed per query. Access SSD is better. More are allowed per query
  - Minimize disk I/O by combing small I/O accesses
- Distributed processing/parallel processing is feasible
  - But watch cost of network communication/latency

Jeff Dean

Numbers Every Engineer Should Know (Approximately)



$ms = 10^{-3}s$

$\mu s = 10^{-6}s$

$ns = 10^{-9}s$

CPU

- L1 cache reference 0.5 ns
- L2 cache reference 7 ns

Memory

- Memory Main memory reference 100 ns
- Read 1 MB sequentially from memory 0.25ms

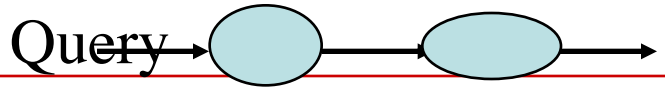
Disk

- HDD Disk seek 8ms while SSD takes 0.1ms for reading 4KB
- Read 1 MB sequentially from disk 10ms-1ms

Network

- Round trip within same datacenter 0.5ms
  - The part of transferring 1K bytes over 1 Gbps network 10μs
- Read 1 MB sequentially from network 10ms
- Send packet CA->Europe->CA 150ms

Modeling of Response Time for Query Processing



Components of time cost

Query response time ≈

+#instruction*CPUCost

+#MemoryAccess*MemoryCost

+#NetworkOPs * NetworkCost

+ #IO-OPs*IOCost

NetworkCost = Startup latency + DataSize/TransferRate

IOCost = Startup latency+ DataSize/TransferRate

What do we learn from these numbers?

Bad for each query

- 1 cache reference 0.5 ns
- L2 cache reference 7 ns
- Memory Main memory reference 100 ns
- Read 1 MB sequentially from memory 0.25ms

Poor L1/L2 locality for compute-intensive core

Scan 1000MB list

- HDD Disk seek 8ms while SSD takes 0.1ms for reading 4KB
- Read 1 MB sequentially from disk 10ms-1ms

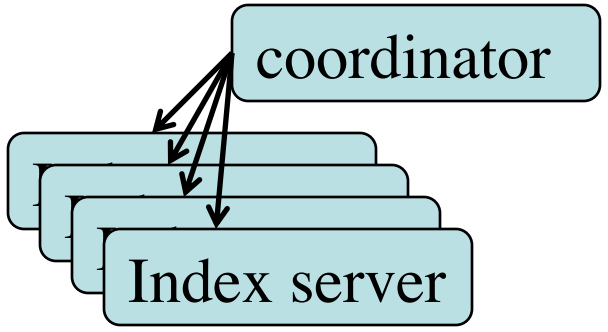
Access disk 10,000 times

- Round trip within same datacenter 0.5ms
  - Transmitting 1K bytes over 1 Gbps network 10μs
- Read 1 MB sequentially from network 10ms
- Send packet CA->Europe->CA 150ms

Remote hash table lookup for 5,000 times

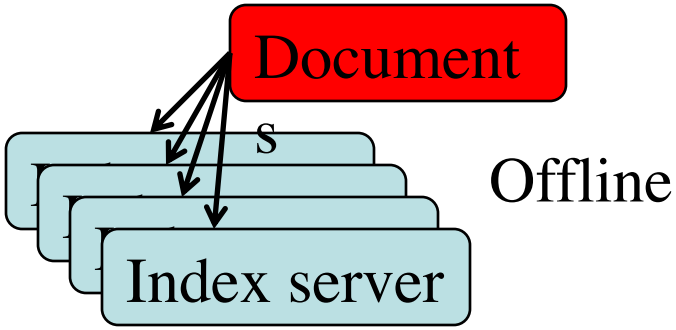
Parallelism Management in a Cluster of Machines for Search

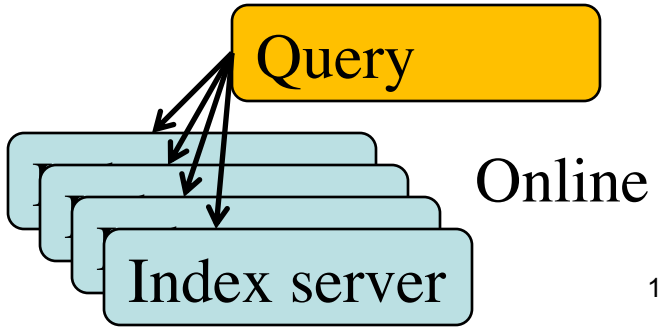
- **Basic steps for parallel processing**
  - All queries sent to a *coordination machine*
  - The coordinator then sends messages to many *index servers*
  - Each index server does some portion of the query processing
  - The coordinator organizes the results and returns them to the user
- **Two main approaches**
  - Document distribution
    - by far the most popular
  - Term distribution



Document-based distribution

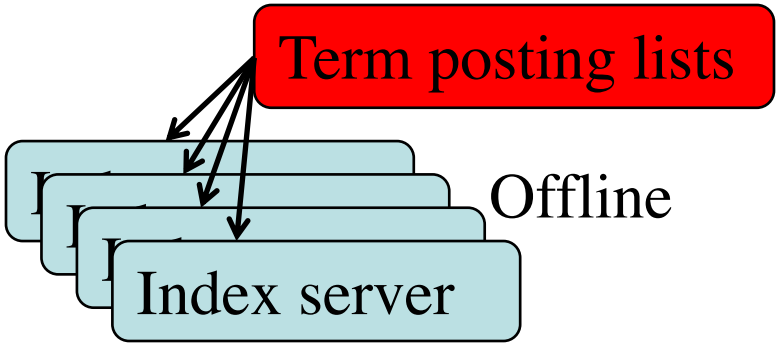
- **Document distribution**
  - Each index server acts as a search engine for a small fraction of the total collection
  - A coordinator sends a copy of the query to each of the index servers, each of which returns the top-*k* results
  - Results are merged into a single ranked list by the coordinator

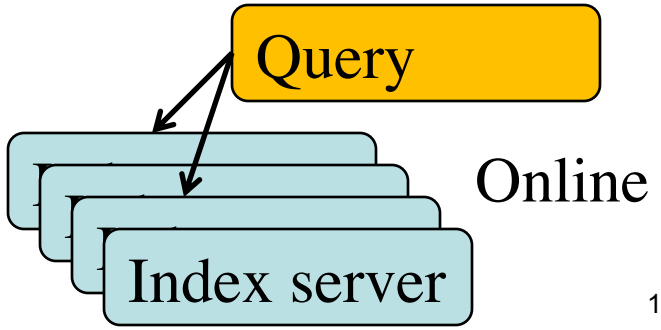




Term-based distribution

- **Single index is built for the entire cluster**
- **Each posting list of a term is assigned to one index server**
- **During query processing,**
  - One of the index servers is chosen to process the query
  - Usually the one holding the longest inverted list
  - Other index servers send information to that server
  - Final results sent to director





Layout of inverted index impacted by online algorithms

- **Early termination of faster query processing**
  - Ignore lower priority documents at end of lists
  - Fast (but unsafe) optimization
- **Ordering of inverted posting lists**
  - Impact sorted index: high score documents first
  - Document sorted index: increasing order of doc IDs
  - How to combine the advantages?

Sort layers by impact, and then sort documents by IDs within each group

Exercise: Design options for fast query processing

Assume 3 word queries	#I/O Operations	Time cost	Design options/strategies
Query word intersection of postings	3 random I/O operations to read 3 posting lists List length upto 100MB	1 or few seconds	
Rank top 1000 results	1000 random I/O operations to access features 1000bytes/doc	1000*HHD access=10 seconds 1000*SSD =100ms	
Generate 10 snippets	10 random I/O operations to read docs Each doc -2KB	10 *HHD =100ms 10*SSD=1ms	

Exercise: Strategies for fast query processing

Assume 3 word queries	#I/O Operations	Time cost	Design options/strategies
Query word intersection of postings	3 random I/O operations to read 3 posting lists. Posting list length upto 100MB	1 or few seconds	• Cache postings • Place the entire index in memory
Rank top 1000 results	1000 random I/O operations to access features 1000bytes/doc	1000*HHD access=10 seconds 1000*SSD =100ms	• Cache features, limited locality • Place all in memory • Use SSD
Generate 10 snippets	10 random I/O operations to read docs Each doc -2KB	10 *HHD =100ms 10*SSD=1ms	Use SSD

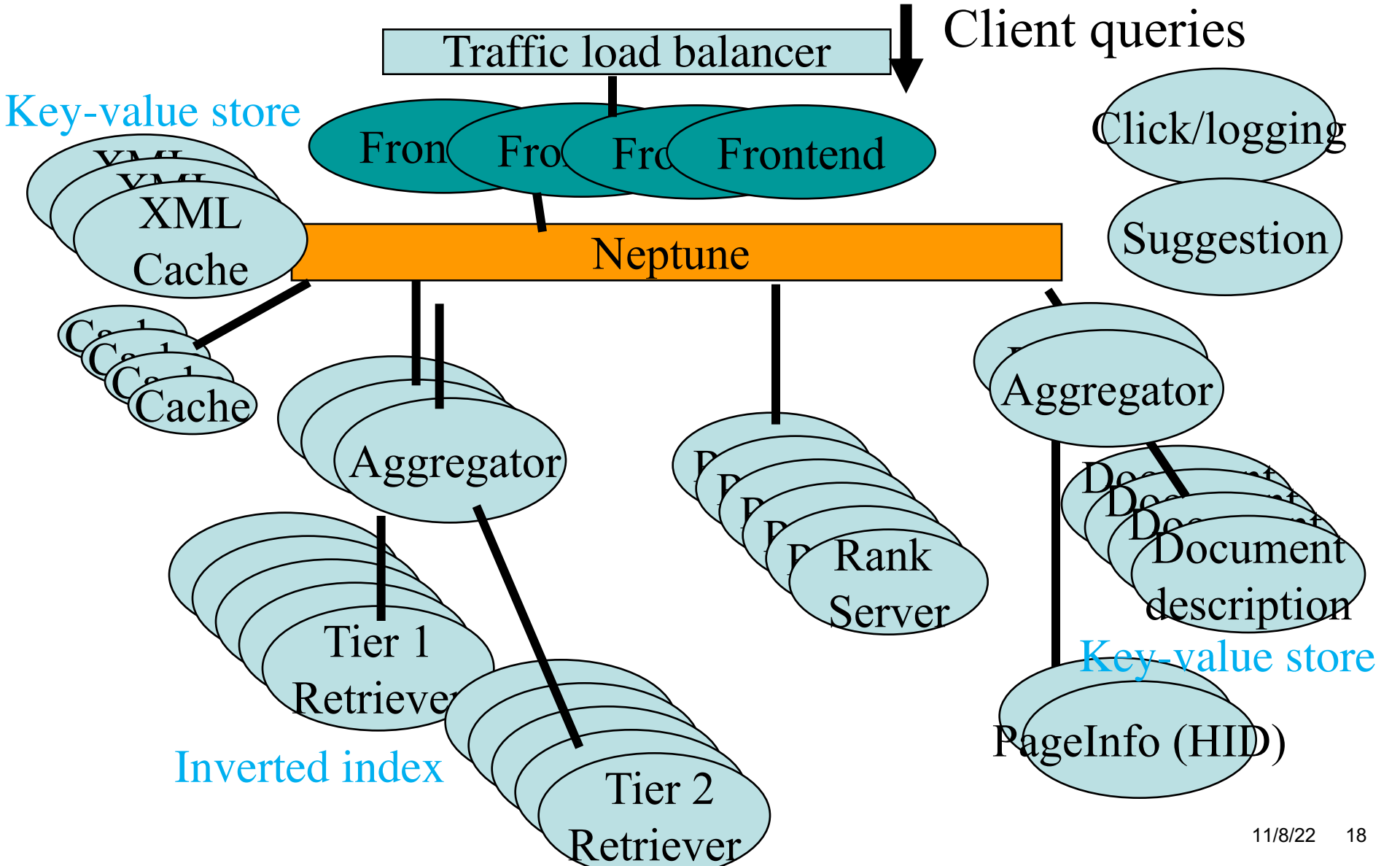
Exercise: Data distribution for parallel computing

Assume p machines for each service	Key datasets and sizes	Method/design options How to assign data to p machines?
Query match n documents m terms	Posting lists of terms Space cost O(n ln m) 2KB/document 100M docs → 200GB	
Rank top K results	Features of documents 100B/document 100M docs→10GB	
Generate 10 snippets	Document text 4KB/document 100M→400GB	

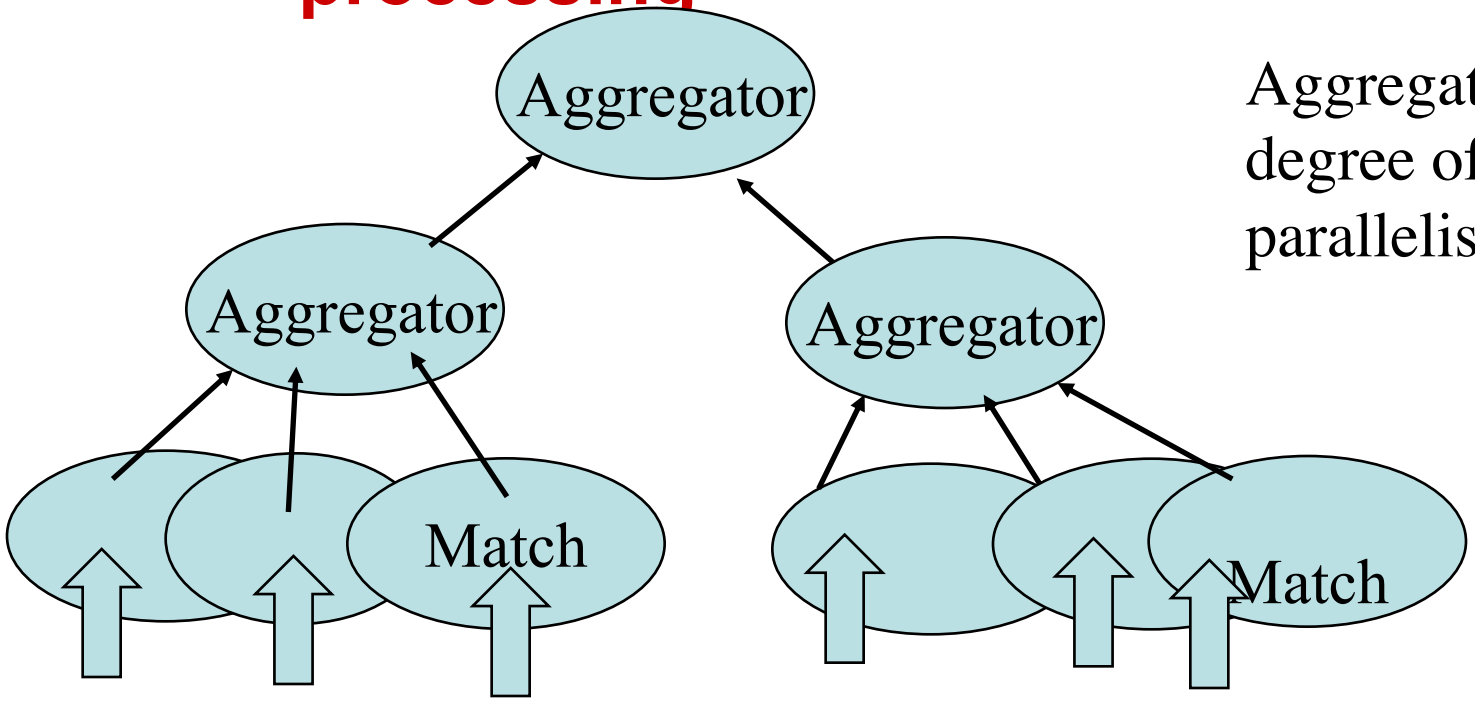
Exercise: Data distribution for parallel computing

Assume p machines for each service	Key datasets and sizes	Design options
Query match	Posting lists of terms 2KB/document 100M docs → 200GB	Document-oriented: Divide/map documents into p machines Term oriented: Divide terms into p machines
Rank top K results	Features of documents 100B/document 100M docs→10GB	Distribute feature vectors by documents to p machines? Or maybe just use one machine
Generate 10 snippets	Document text 4KB/document 100M→400GB	Distribute documents to p machines

Ask.com Search Engine

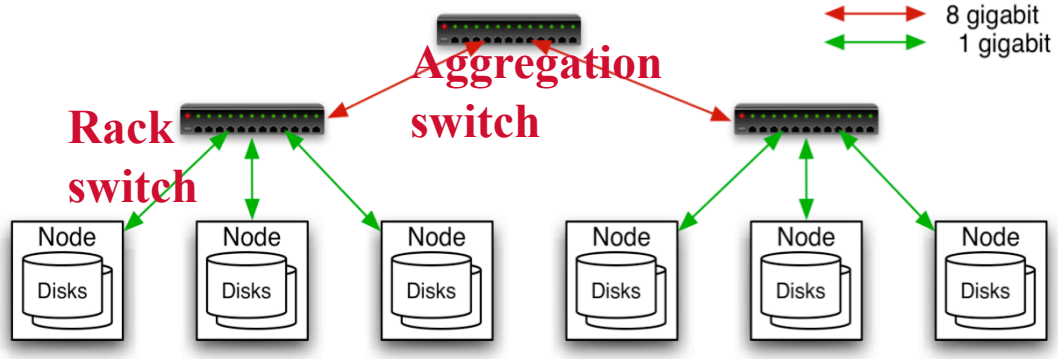


Multi-tier aggregation for query stream processing

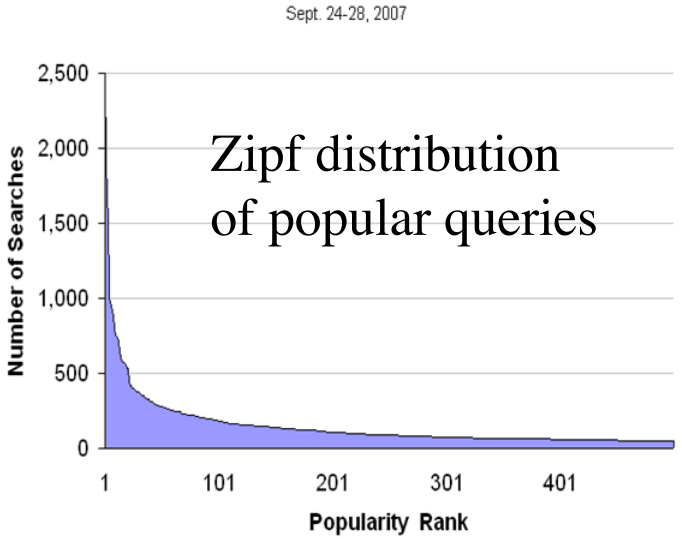


Aggregator controls degree of fan-out parallelism

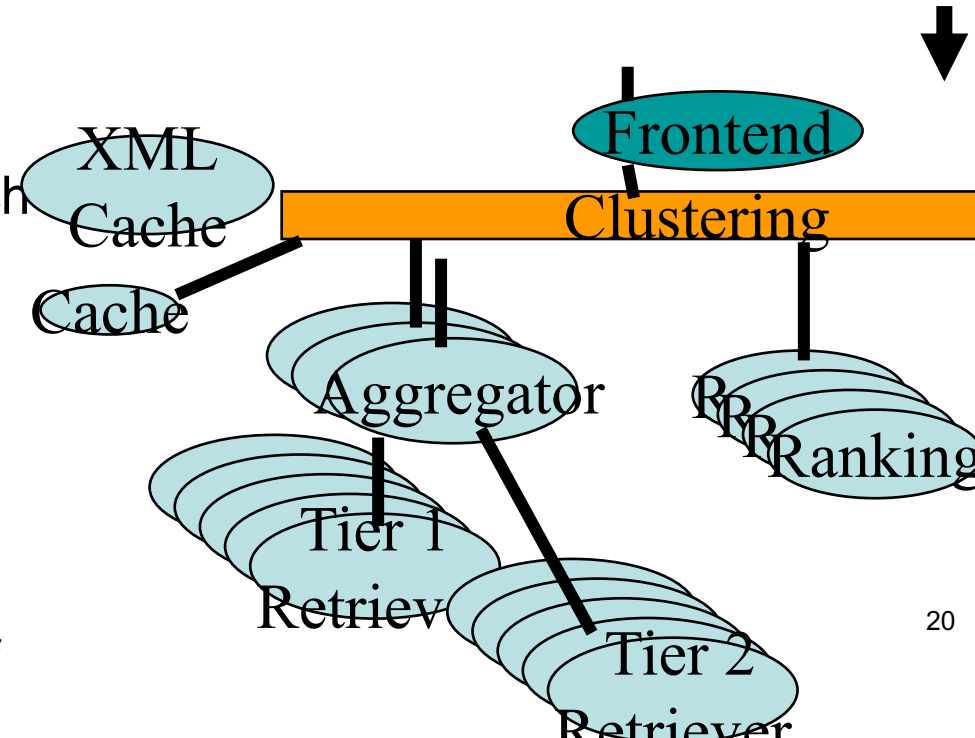
Cluster architecture



Online Architecture: Frontends and Cache

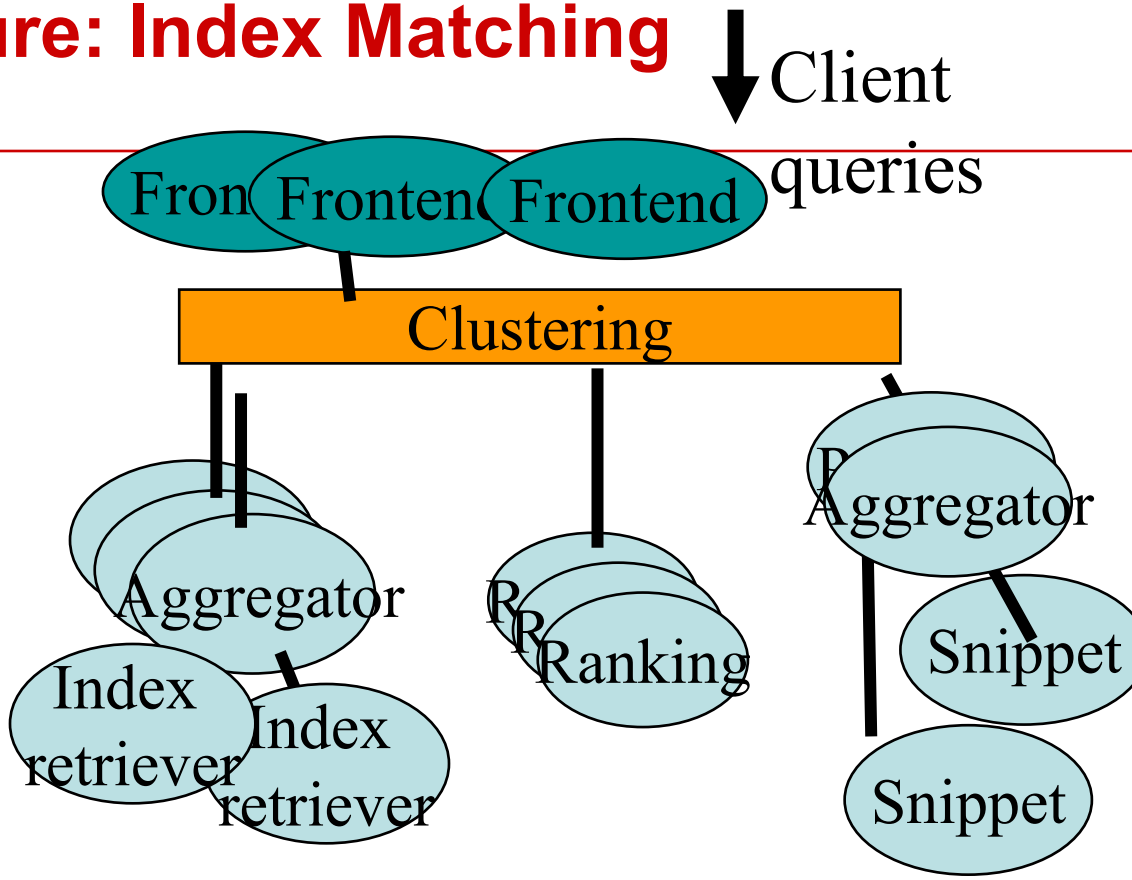


- **Front-ends**
  - Receive web queries
  - Spawn a thread to handle a request
    - Use cache if possible
    - Otherwise call index matching/ranking
  - Then present results to clients (XML).
- **XML cache :**
  - Save previously-answered search results (dynamic Web content).
  - Use these results to answer new queries.
- **Result cache**
  - Contain all matched URLs for a query.
    - It does not contain the description of these URLs



Online Architecture: Index Matching and Ranking

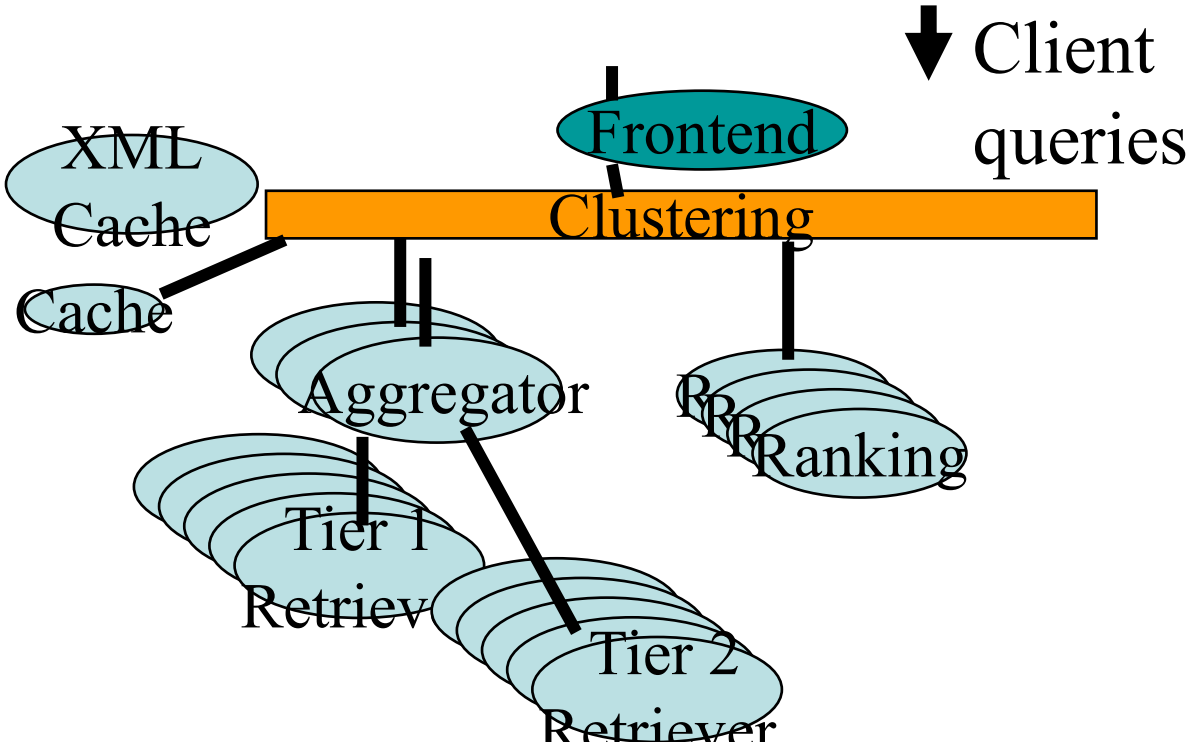
- **Retriever aggregators** (Index match coordinator)
  - Gather results from online database partitions.
- **Index retrievers**
  - Match pages relevant to query keywords
- **Ranking server**
  - Classify pages into topics & Rank pages



- **Snippet aggregators**
  - Combine descriptions of URLs
- **Dynamic snippet servers**
  - Extract proper description for a given URL.

Distributed Coordination on Service Availability

- **Making a remote service call is possible, but how to coordinate information sharing?**
  - How does a machine know there are multiple copies of the same remote service available?
  - How does a machine know a remote service is down?



The Neptune Clustering Middleware

Programming challenges/requirements for online services:

Data intensive, requiring large-scale clusters.

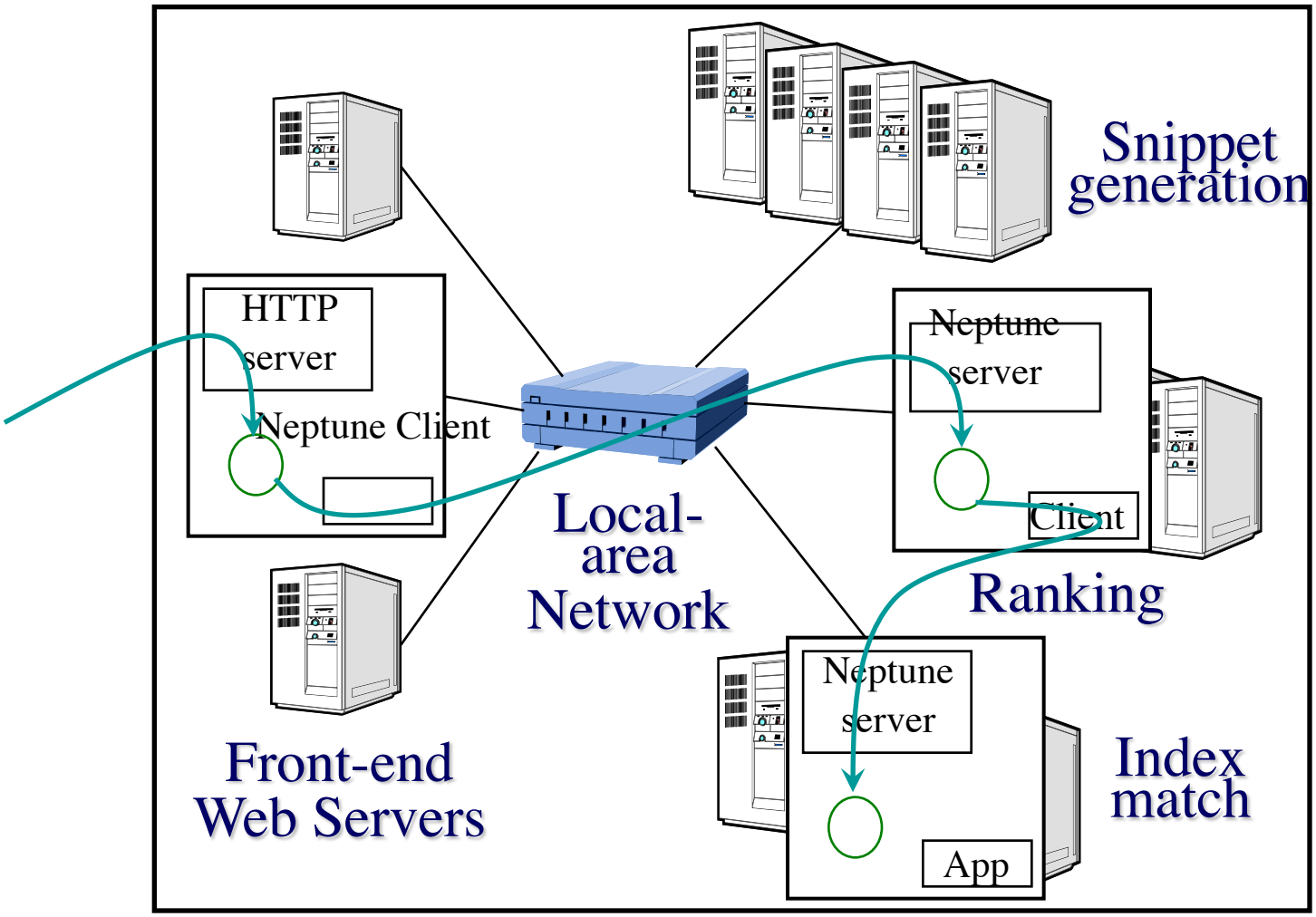
Incremental scalability. 7×24 availability.

Resource management, QoS for load spikes.

Lack of programming support for reliable/scalable online network services and applications.

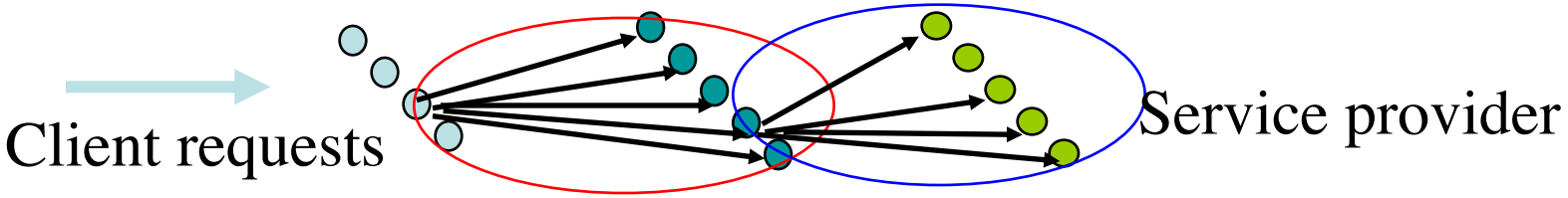
- Neptune: Clustering middleware for aggregating and replicating application modules with persistent data.
- A simple and flexible programming model to shield complexity of service discovery, load scheduling, consistency, and failover management
- www.cs.ucsb.edu/projects/neptune for code, papers, documents.
  - K. Shen, et. al, OSDI 2002. PPoPP 2003.

Example: a Neptune Clustered Service: Index match service



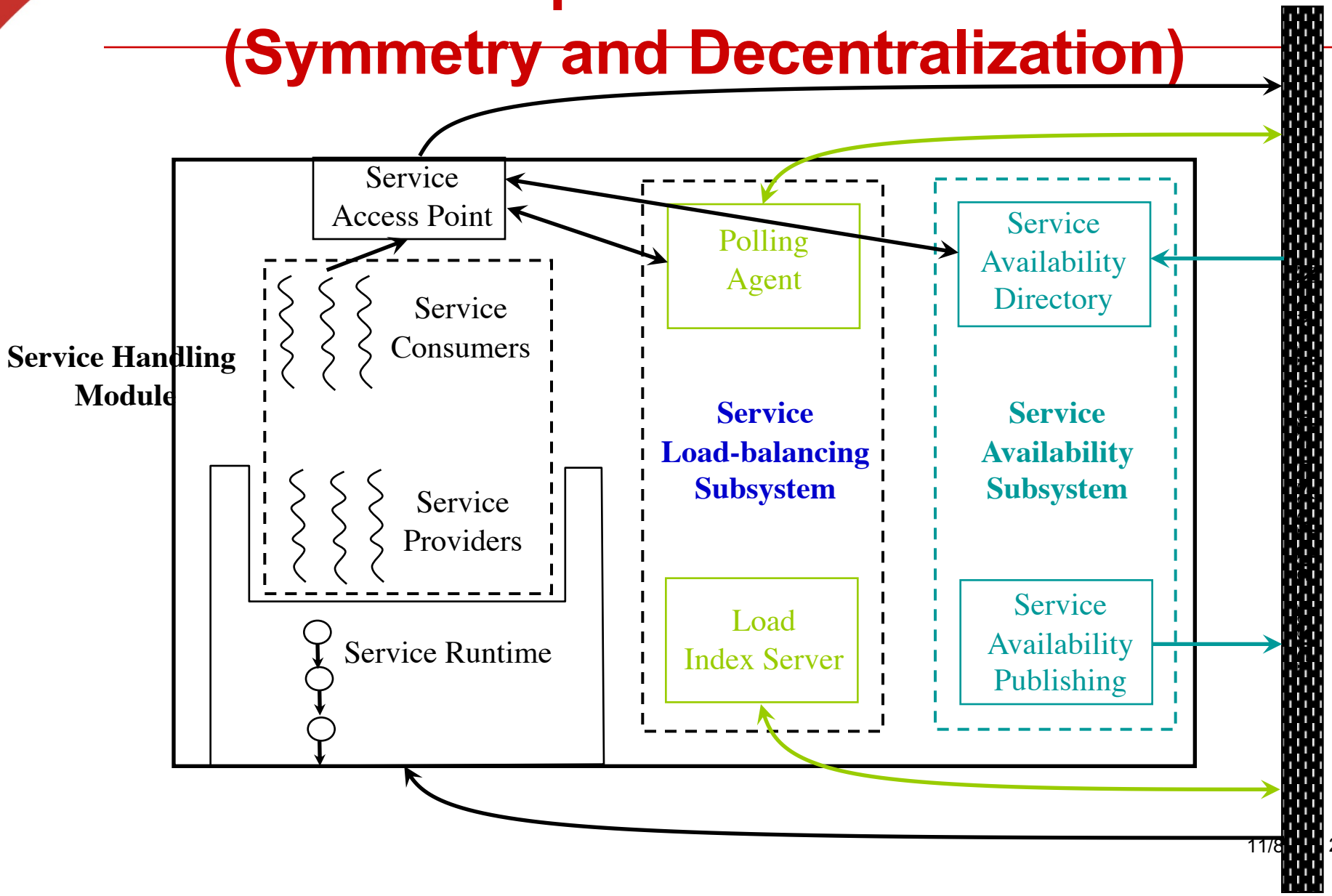
Neptune architecture for cluster-based services

- **Symmetric and decentralized:**
  - Each node can host multiple services, acting as a service provider (Server)
  - Each node can also subscribe internal services from other nodes, acting as a consumer (Client)
    - *Advantage: Support multi-tier or nested service architecture*



- **Neptune components at each node:**
  - Application service handling subsystem.
  - Load balancing subsystem.
  - Service availability subsystem.

Inside a Neptune Server Node (Symmetry and Decentralization)

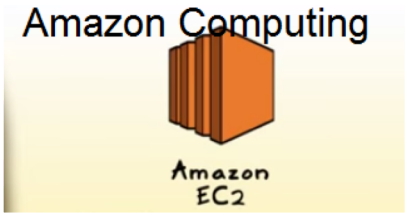


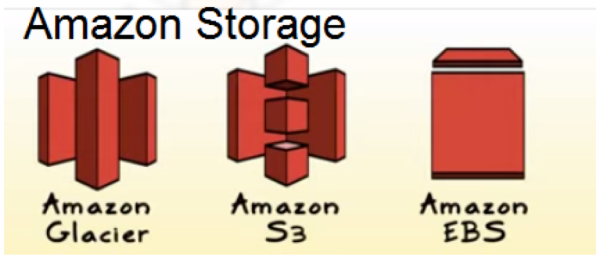
Availability and Load Balancing

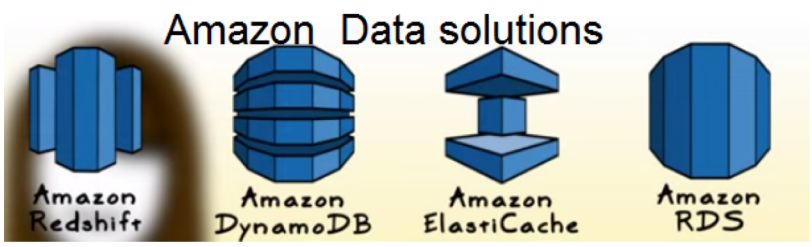
- **Availability subsystem**:
  - Announcement once per second through IP multicast;
  - Availability info kept as soft state, expiring in 5 seconds;
  - Service availability directory kept in shared-memory for efficient local lookup.
- **Load-balancing subsystem**:
  - Challenging: medium/fine-grained requests.
  - Random polling with sampling.
  - Discarding slow-responding polls

Online/Offline Processing on the Cloud

- **Building an online/offline service on the cloud can be cost-effective**
  - Many services run on AWS, Microsoft Azure, Google Cloud
  - Netflix is built with AWS and runs on AWS
- **AWS (Amazon Web Services)** is a programmable collection of remote computing services as building blocks for development and deployment:
  - Elastic Compute Cloud (EC2): virtual private servers using Xen.
  - SimpleDB: query processing with core functionality of a database
  - DynamoDB – key-value stores
  - Elastic search – document search with inverted index
  - AElastic MapReduce supports Map Reduce programming with Hadoop
  - Much more



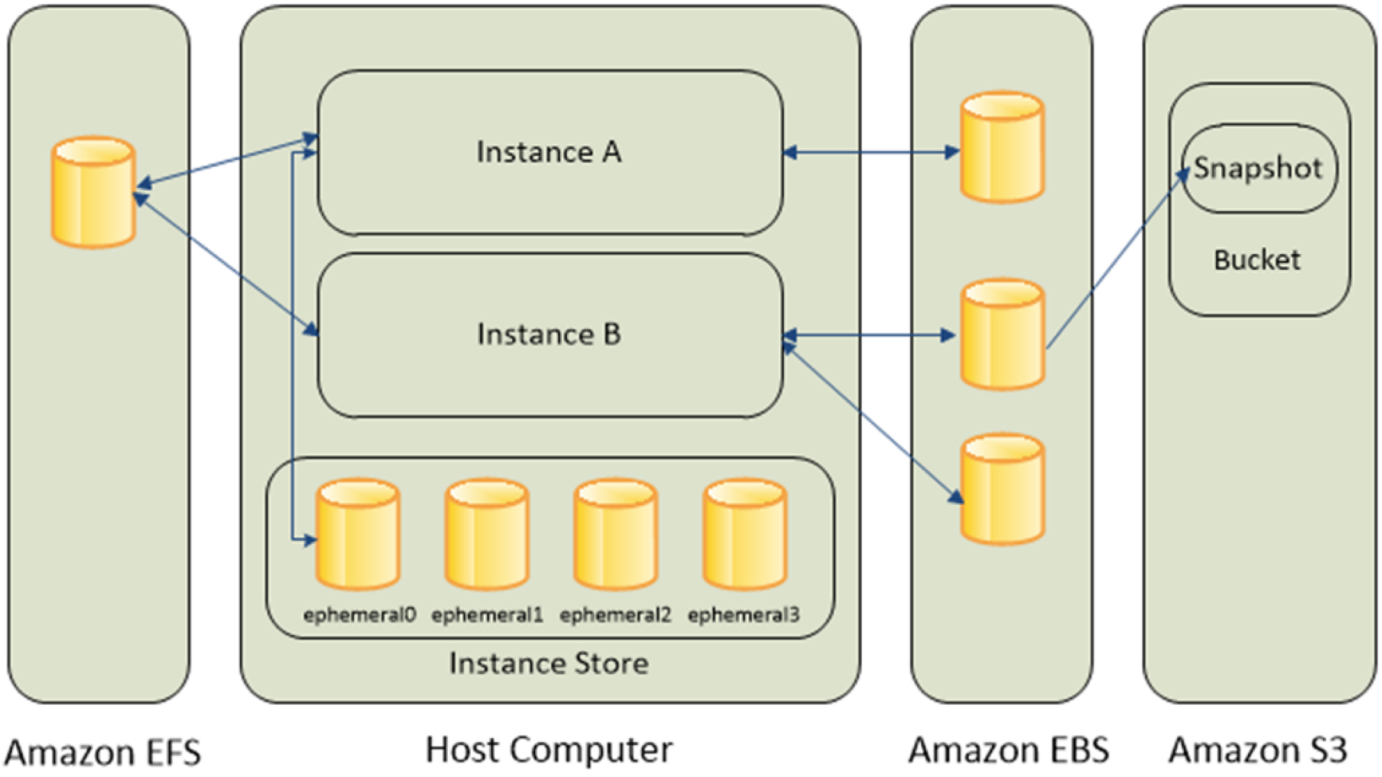




Amazon Elastic Compute Cloud (EC2)

- An EC2 instance appears as physical HW, provides users full control over nearly entire sw stack, from the kernel upwards
- On-Demand: Pay for capacity without long-term commitment

The following figure shows the relationship between these storage options and your instance.



Storage Options for EC2

1. Amazon EBS

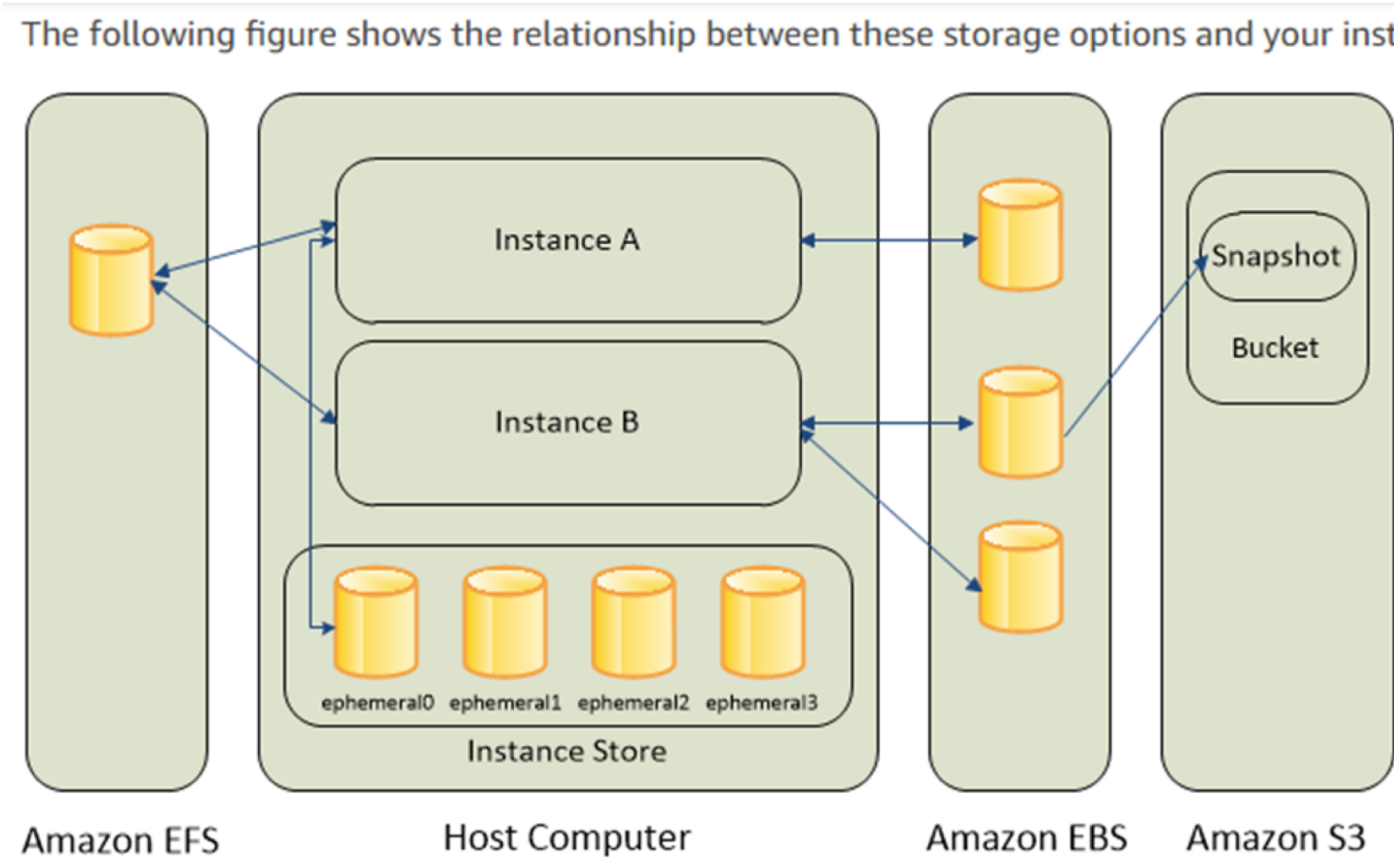
Essentially hard disks for the majority of use cases with persistent and block-level storage (e.g. useful for a database or file system)

2. Instance Store

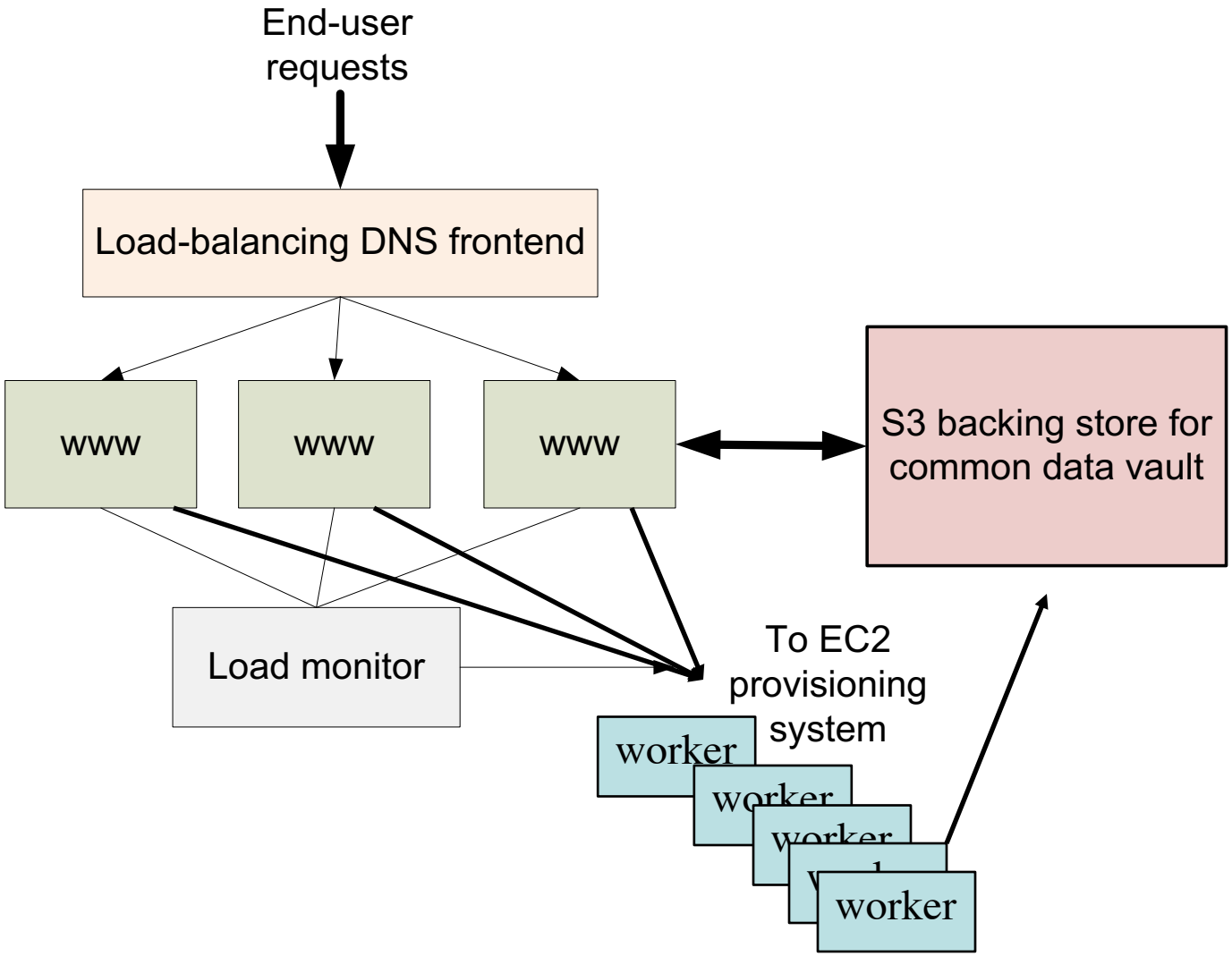
Inexpensive temporary storage. This doesn't persist if the instance is stopped or terminated.

3. Amazon S3

An internet storage with a simple web service interface for storage and backup.



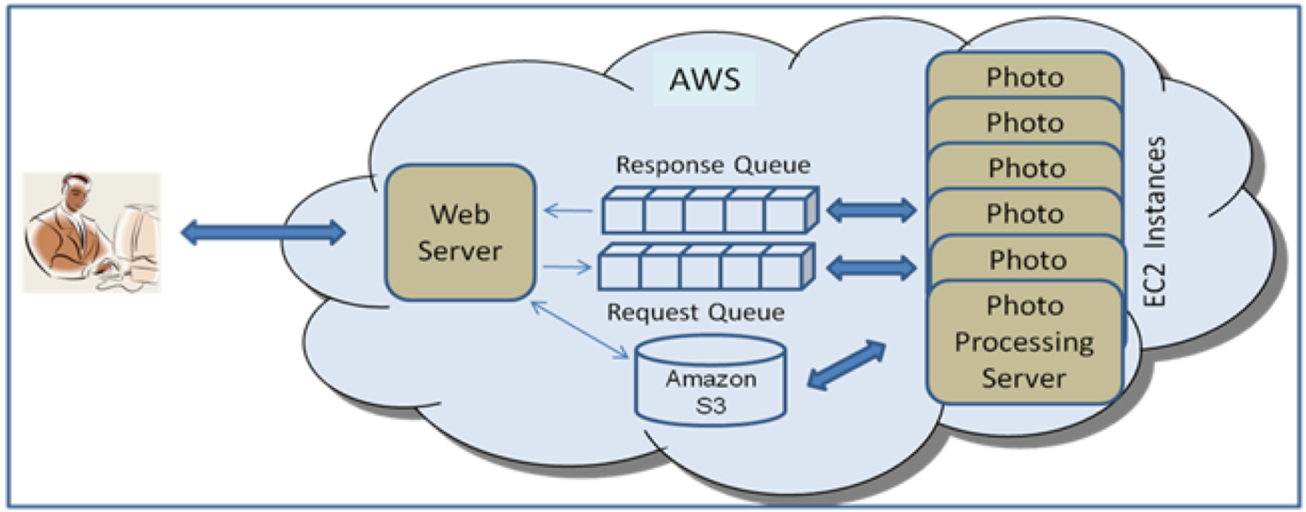
Self-Scaling Online Applications with AWS



Examples of building blocks for online query processing

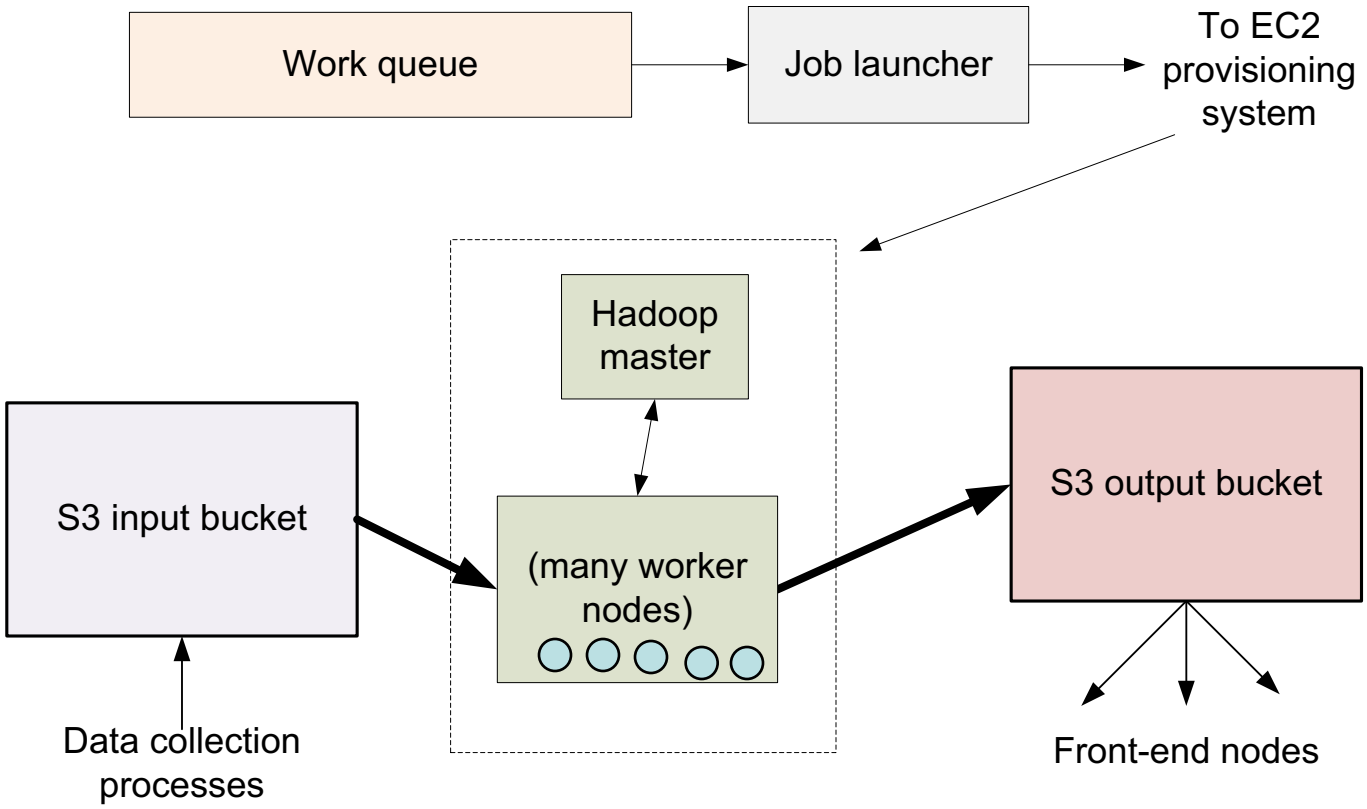
Example: Online photo processing service

- **Photo operation**
  - Red eye reduction/cropping/customization/re-coloring/teeth whitening, etc
- **Architecture design with AWS**
  - Web server receive request
  - Put request message in the queue
  - Pictures stored in S3
  - Multiple EC2 instances run photo processing
  - Put back in the queue
  - Return



Self-Scaling Long-Job Processing with AWS

Example building blocks for handling long jobs such as offline data processing using Hadoop



Case study: Grep the web

- Crawling the web
- Large web crawl data is stored in S3
- Users can submit regular expression to the "search" program – "**Grep the web**"
  - uses Hadoop to search for data
  - Puts your results in an output bucke and notifies you when it's ready

Figure 3: Phases of GrepTheWeb Architecture

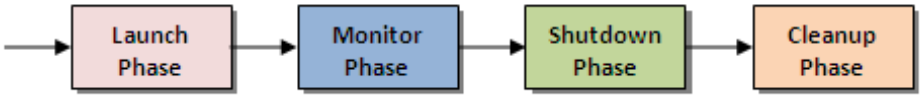
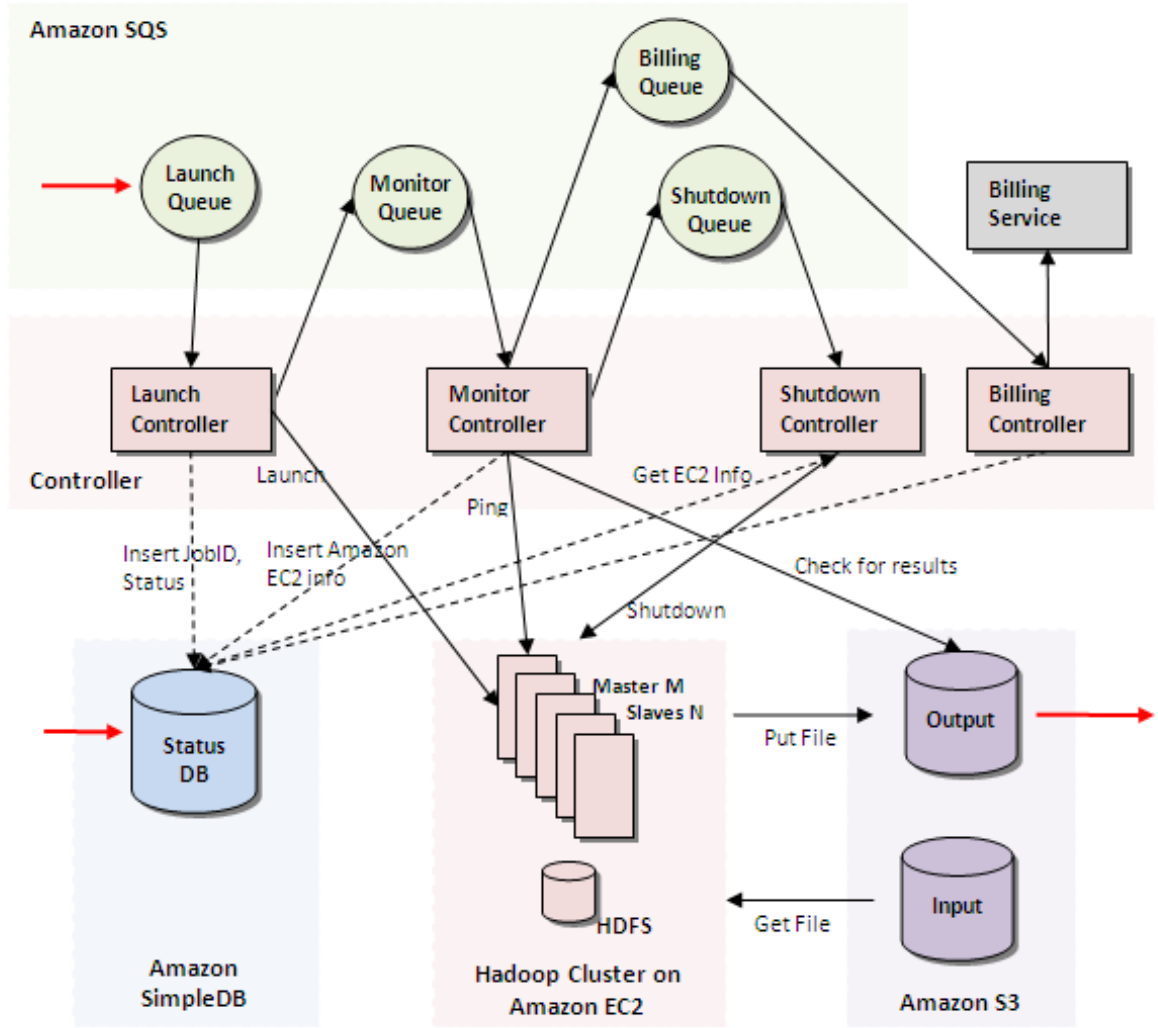
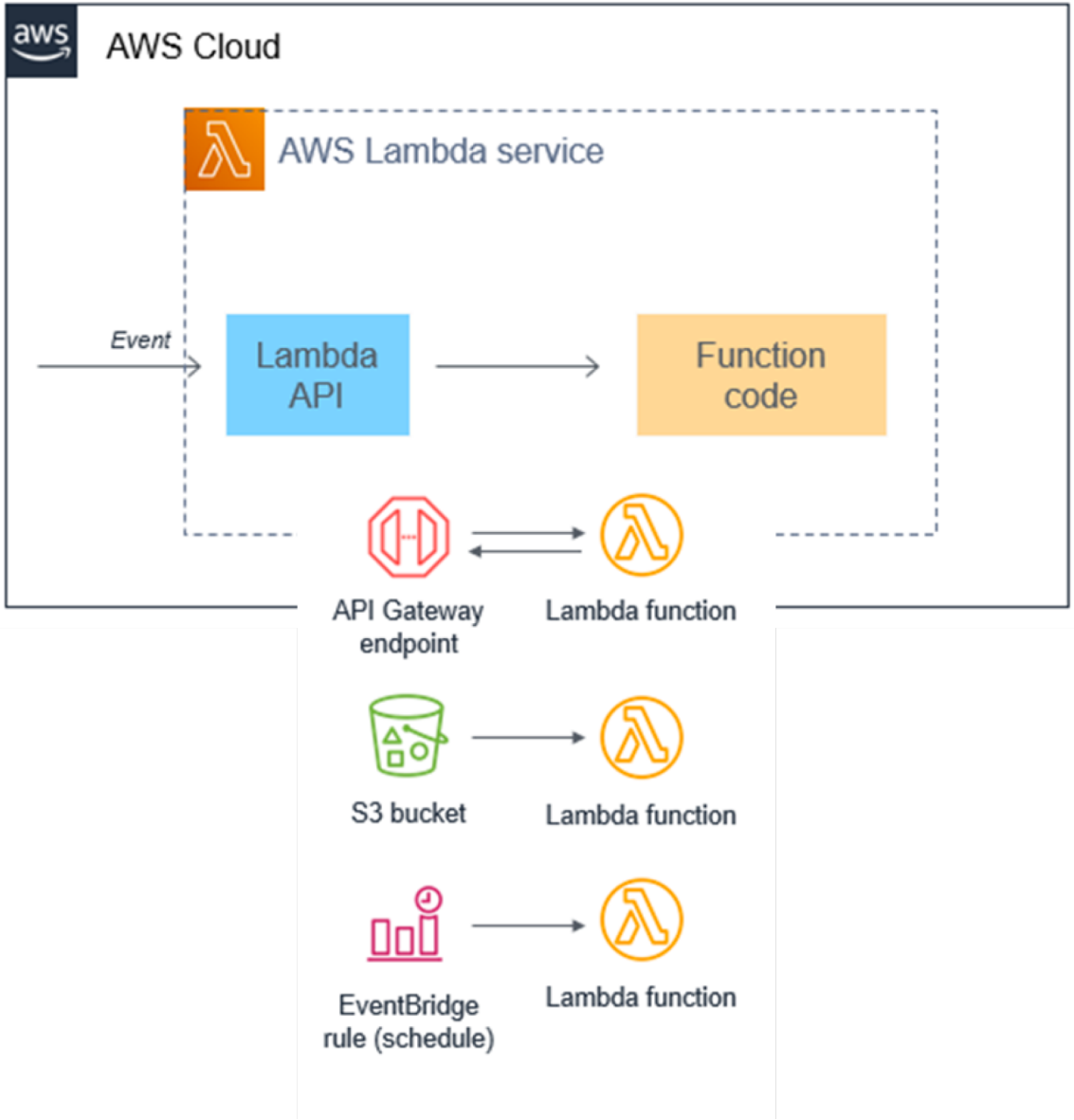


Figure 4: GrepTheWeb Architecture - Zoom Level 3



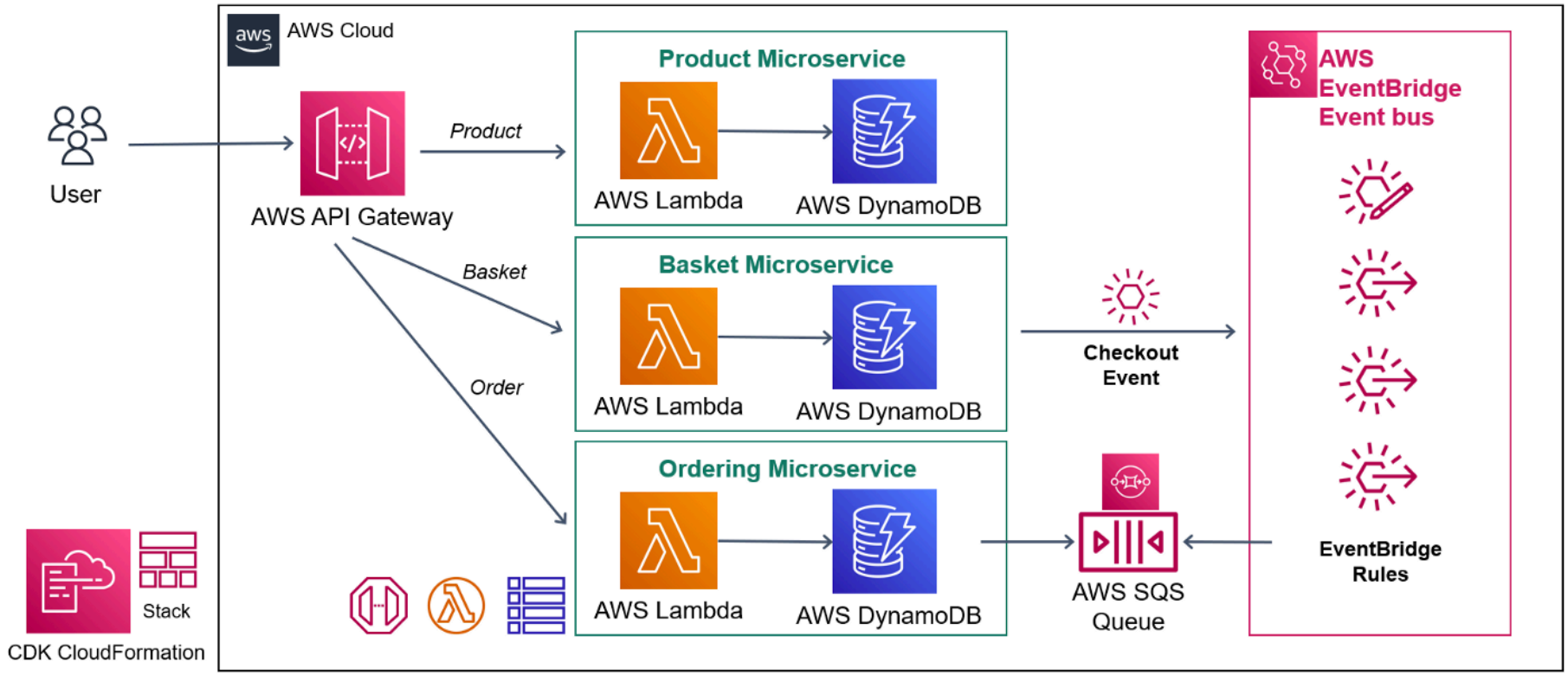
AWS Lambda

- **Lambda** is a **compute service** that runs custom function code and return response to events.
- **Serverless event-driven microservices** No need to provision or manage servers
- Most AWS services generate events for communicating each other. They are event sources for Lambda.



AWS Lambda: Event-driven Architecture

- AWS Lambda can be used for **serverless event-driven microservices** such as e-commerce applications

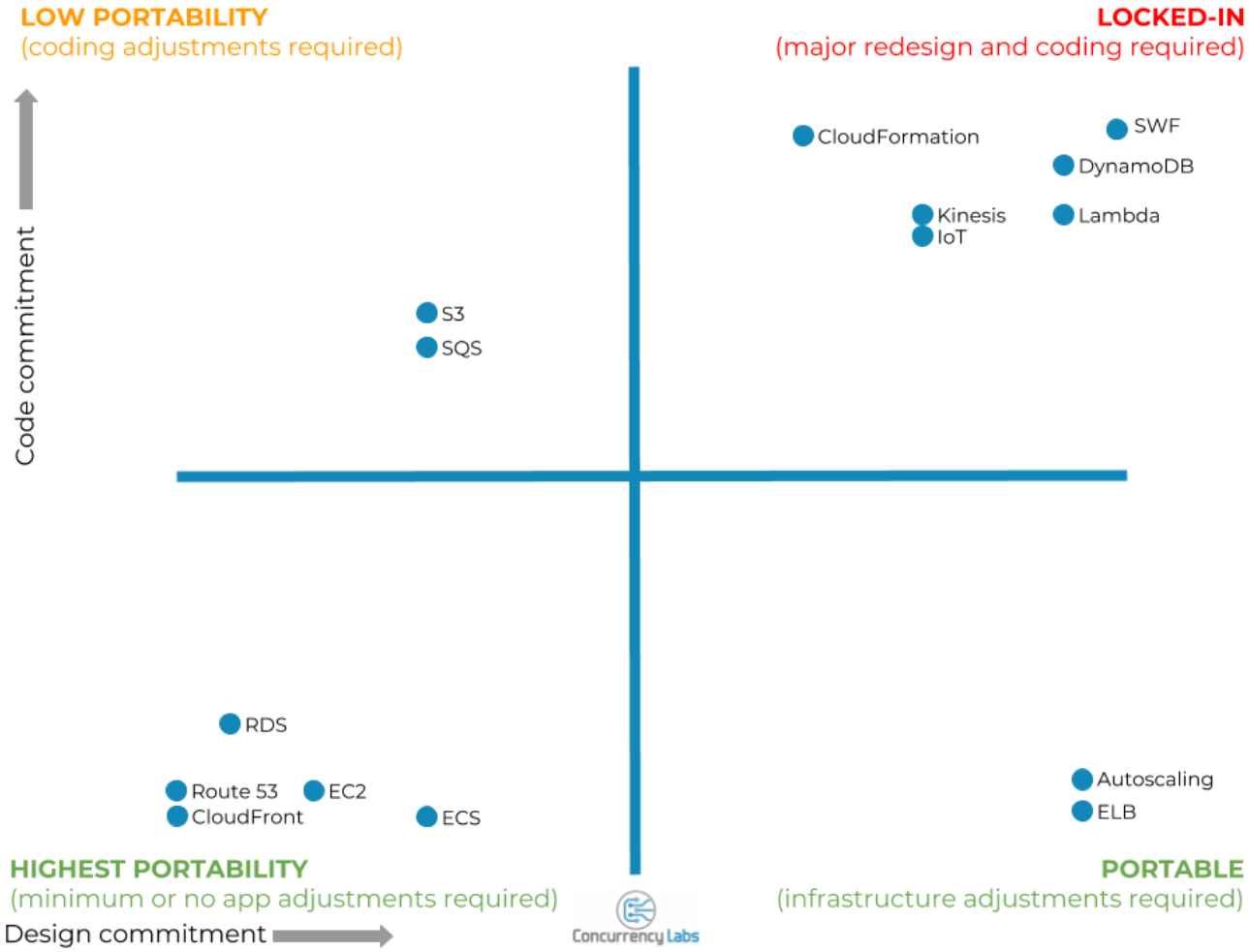


Considerations in choosing a web service

- AWS has over 100 services as building blocks. What to consider?
- Functionality: operations supported.
- Service limit:
  - Number of provisioned resources available
  - Access latency: typical response time
  - Throughput: # concurrent Lambda executions, EFS read/write speed Gb/s)
  - Payload size (Queue message size, Key-value DB item size)
  - Storage requirement or serverless
- Cost of services. Autoscaling vs. manual scaling
- Availability degree, redundancy options, monitoring tools
- Supported programming languages
- Easy degree to integration with other services

Portability of design and code in using cloud services

- Some services can be used without doing any design or code updates. But some requires significant changes



AWS

Takeaways for Online Query Processing

- **A complex system can use tens or hundreds of services running on thousands of machines**
- **Performance consideration in algorithm/architecture**
  - Data scalability: what happens data size increases by 100x Also software/machine/human scalability
  - Interaction of response time and throughput with high traffic
- **Strategies for faster performance**
  - Caching in memory hierarchy
  - Parallel processing of query matching and ranking
    - Different distribution: Documents vs terms
- **Building/running services on the cloud (e.g. AWS) is popular**
  - Elastic computing
  - Software as a service (SaaS)