## Query Processing and Online Architectures

•T. Yang 290N 2013

•Partially from Croft, Metzler & Strohman's textbook

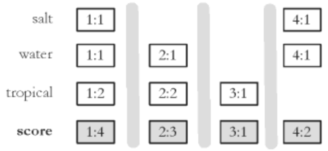
## Content

- **Query processing flow and data distribution.**
- **Experience with Ask.com online architecture**
  - Service programming with Neptune.
  - Zookeeper

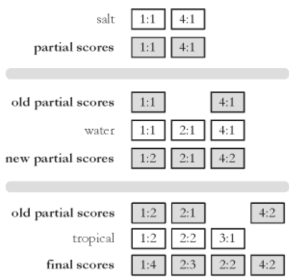
## Query Processing

- **Document-at-a-time**
  - Calculates complete scores for documents by processing all term lists, one document at a time
- **Term-at-a-time**
  - Accumulates scores for documents by processing term lists one at a time
- **Both approaches have optimization techniques that significantly reduce time required to generate scores**

## Document-At-A-Time



## Term-At-A-Time



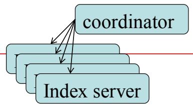

## Optimization Techniques

- **Term-at-a-time uses more memory for accumulators, data access is more efficient**
- **Optimization**
  - Read less data from inverted lists
    - e.g., skip lists
    - better for simple feature functions
  - Calculate scores for fewer documents
  - Threshold-based elimination
    - Avoid to select documents with a low score when high-score documents are available.

## Other Approaches

- **Early termination of query processing**
  - ignore high-frequency word lists in term-at-a-time
  - ignore documents at end of lists in doc-at-a-time
  - *unsafe* optimization
- **List ordering**
  - order inverted lists by quality metric (e.g., PageRank) or by partial score
  - makes unsafe (and fast) optimizations more likely to produce good documents

## Distributed Evaluation




- **Basic process**
  - 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

## Distributed Evaluation

- **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

## Distributed Evaluation

- **Term distribution**
  - Single index is built for the whole cluster of machines
  - Each inverted list in that index is then assigned to one index server
    - in most cases the data to process a query is not stored on a single machine
  - 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

## Caching

- **Query distributions similar to Zipf**
  - Over 50% of queries repeat → cache hit
  - Some hot queries are very popular.
- **Caching can significantly improve response time**
  - Cache popular query results
  - Cache common inverted lists
- **Inverted list caching can help with unique queries**
- **Cache must be refreshed to prevent stale data**

## Open-Source Search Engines

- **Apache Solr:** http://lucene.apache.org/solr/
  - full-text search with highlighting, faceted search, dynamic clustering, database integration, rich document (e.g., Word, PDF) handling, and geospatial search
  - distributed search and index replication.
  - Based on Java Apache Lucene search.
- **Constellio**: http://www.constellio.com/

Open-source enterprise level search based on Solr.

- **Zoie:** sna-projects.com/zoie/ – Real time search indexing built ontop of Lucene.

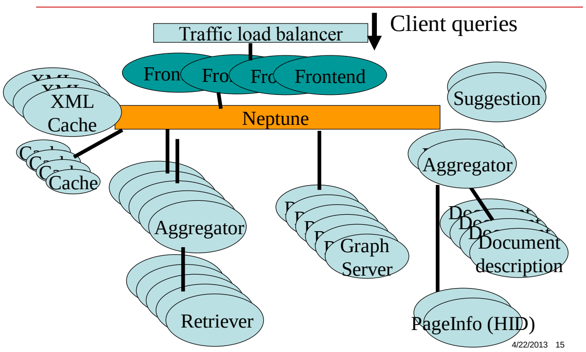
## Open-Source Search Engines

- **Lemur** **http://www.lemurproject.org/**
  - C/C++, running on Linux/Mac and windows.
  - Indri search engine by U. Mass/CMU.
  - Parses PDF, HTML, XML, and TREC documents. Word and PowerPoint parsing (Windows only).
  - UTF-8
- **Sphinx:** http://sphinxsearch.com/
  - Cross platform open source search server written in C++
  - search across various systems, including database servers and NoSQL storage and flat files.
- **Xapian**: xapian.org/ – search library built on C++

## Fee-based Search Solutions

- **Google SiteSearch** http://www.google.com/sitesearch/
  - Site search is aimed primarily at websites, and not for an intranet.
  - It is a fully hosted solution
  - Pricing for site search is on a query basis per year. Starting at $100 for 20,000 queries a year
- **Google Mini**
  - a server based solutions. Once deployed, Mini crawls your Web sites *and* file systems / internal databases,
  - Costs start at $1,995 (direct) plus a $995 yearly fee after the first year for indexing of 50,000 documents, and scales upwards

## Ask.com Search Engine






## Frontends and Cache

- **Front-ends**
  - Receive web queries.
  - Direct queries through XML cache, compressed result cache, database retriever aggregators, page clustering/ranking,
  - Then present results to clients (XML).
- **XML cache :**
  - Save previously-queried search results (dynamic Web content).
  - Use these results to answer new queries. Speedup result computation by avoiding content regeneration
- **Result cache**
  - Contain all matched URLs for a query.
  - Given a query, find desired part of saved results. Frontends need to fetch description for each URL to compose the final XML result.

## Index Matching and Ranking

- **Retriever aggregators (Index match coordinator)**
  - Gather results from online database partitions.
  - Select proper partitions for different customers.
- **Index database retrievers**
  - Locate pages relevant to query keywords.
  - Select popular and relevant pages first.
  - Database can be divided as many content units
- **Ranking server**
  - Classify pages into topics & Rank pages
- **Snippet aggregators**
  - Combine descriptions of URLs from different description servers.
- **Dynamic snippet servers**
  - Extract proper description for a given URL.



## Programming Challenges for Online Services

- **Challenges/requirements for online services:**
  - Data intensive, requiring large-scale clusters.
  - Incremental scalability.
  - 7×24 availability.
  - Resource management, QoS for load spikes.
- **Fault Tolerance:**
  - Operation errors
  - Software bugs
  - Hardware failures
- **Lack of programming support for reliable/scalable online network services and applications.**

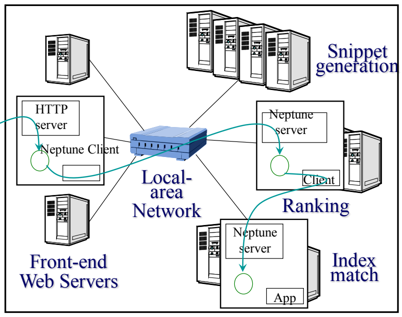


## The Neptune Clustering Middleware

- 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, USENIX Symposium on Internet Technologies and Systems, 2001

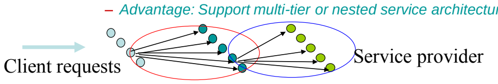


## Example: a Neptune Clustered Service: Index match service



## Neptune architecture for cluster-based services

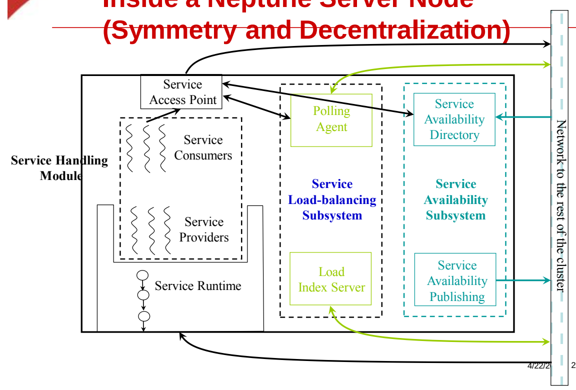
- **Symmetric and decentralized:**
  - Each node can host multiple services, acting as a service provider (Server)
  - Each node can also subscribe internal services from other nodes, acting as a consumer (Client)
    - *Advantage: Support multi-tier or nested service architecture*




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



## Inside a Neptune Server Node (Symmetry and Decentralization)





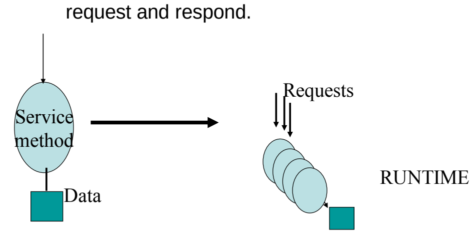

## Availability and Load Balancing

- **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



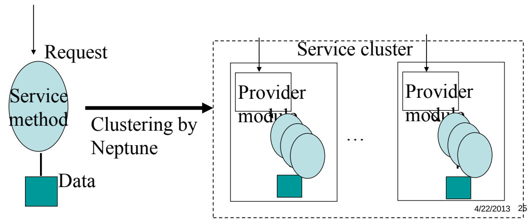
## Programming Model in Neptune

- **Request-driven processing model:** programmers specify service methods to process each request.
- **Application-level concurrency:** Each service provider uses a thread or a process to handle a new request and respond.



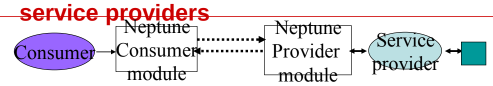
## Cluster-level Parallelism/Redudancy

- **Large data sets** can be partitioned and replicated.
- **SPMD model** (single program/multiple data).
- **Transparent service access:** Neptune provides runtime modules for service location and consistency.






## Service invocation from consumers to service providers




- **Request/response messages:**
  - *Consumer side:* NeptuneCall(service_name, partition_ID, service_method, request_msg, response_msg);
  - *Provider side:* “service_method” is a library function. Service_method(partitionID, request_msg, result_msg);
  - *Parallel invocation with aggregation*
- **Stream-based communication:** Neptune sets up a bi-directional stream between a consumer and a service provider. Application invocation uses it for socket communication.



## Code Example of Consumer Program

Hp=NeptuneInitClt(LogFile);

NeptuneConnect (Hp, “IndexMatch”, 0,
 Neptune_MODE_READ, “IndexMatchSvc”, &fd, NULL);

…Then use fd to read/write data…

NeptuneFinalClt(Hp);



## Example of server-side API with stream-based communication

- **Server-side functions**

Void IndexMatchInit(Handle)
 Initialization routine.

Void IndexMatchFinal(Handle)
 Final processing routine.

Void IndexMatchSvc(Handle, paritiitionID, ConnSd)
 Processing routine for each indexMatch request.

## Publishing Index Search Service

- **Example of configuration file**

**[IndexMatch]**
SVC_DLL = /export/home/neptune/IndexTier2.so
LOCAL_PARTITION = 0,4          # Partitions hosted
INITPROC=IndexMatchInit
FINALPROC=IndexMatchFinal
STREAMPROC=IndexMatchSvc



## ZooKeeper

- **Coordinating distributed systems as “zoo” management**
  - http://zookeeper.apache.org
- **Open source high-performance coordination service for distributed applications**
  - Naming
  - Configuration management
  - Synchronization
  - Group services