Search Engine Architecture

1. Introduction
What is this course about?

• Information retrieval on big data
• Broad survey of system architectures that enable big data applications
• In-depth focus on key insights behind each
  • Without getting too bogged down
IR is Everywhere

- Web search (Google, Bing, ...)
- Finance (Bloomberg)
- Advertising (AdSense, ...)
- Fraud detection
- Medical diagnostics
- What are the major architectural problems?
Big Data Storage

- KV – S3, Cassandra, Riak, Redis
- Document – MongoDB, CouchDB, Elasticsearch
- Column – VoltDB, Vertica
- Graph – Neo4j, OrientDB
- Wide Column – BigTable, HBase
Big Data Processing

- MapReduce – Hadoop, Hive, Pig
- DAG – Storm, Dryad, Spark
- Vertex-centric – Pregel, GraphLab
Course
Administrivia
Prerequisites

- CSCI-GA 1170 Fundamental Algorithms
- CSCI-GA 2110 Programming Languages
- CSCI-GA 2250 Operating Systems
- Working knowledge of Python
- Ability to Google solutions to problems as they come up
Details

• Lectures Wed 5:10-7pm
• Office hours after class
• Mailing list
  • See assignment 1 for the link if you’re not sure
• Grading
  • 10% Class Participation
  • 50% Assignments
  • 40% Final project
Details

• Readings
  • All available online – see course website
  • *Introduction to Information Retrieval* by Manning et al.
  • *Data-Intensive Text-Processing with MapReduce* by Lin et al.
  • Relevant academic articles
Assignments

• Designed to introduce you to the material in a structured way

• Feel free to work together, but everything submitted must be prepared (typed) individually

• Late policy
  • Up to 24 hours late: 0.75 multiplier
  • 24 to 48 hours late: 0.5 multiplier
  • And so on
Assignments

• In the history of CS, these topics are extremely new
• New things tend to break
  • Bugs, missing/wrong documentation, incompatible versions
• Be patient
  • We will inevitably encounter problems
• Be flexible
  • We will find workarounds
• Be constructive
  • Tell me how I can improve everyone’s experience
Tackling Big Data
Divide and Conquer

Parallelization Challenges

• How do we assign work units to workers?
• What if we have more work units than workers?
• What if workers need to share partial results?
• How do we aggregate partial results?
• How do we know all the workers have finished?
• What if workers die?

What’s the common theme of all of these problems?

Common Theme?

- Parallelization problems arise from:
  - Communication between workers (e.g., to exchange state)
  - Access to shared resources (e.g., data)
- Thus, we need a synchronization mechanism

Managing Multiple Workers

• Difficult because
  • We don’t know the order in which workers run
  • We don’t know when workers interrupt each other
  • We don’t know when workers need to communicate partial results
  • We don’t know the order in which workers access shared data

• Thus, we need:
  • Semaphores (lock, unlock)
  • Conditional variables (wait, notify, broadcast)
  • Barriers

• Still, lots of problems:
  • Deadlock, livelock, race conditions...
  • Dining philosophers, sleeping barbers, cigarette smokers...

• Moral of the story: be careful!

Traditional Tools

• Programming models
  • Shared memory (pthreads)
  • Message passing (MPI)

• Design Patterns
  • Master-slaves
  • Producer-consumer flows
  • Shared work queues

Concurrency Realities

- Concurrency is difficult to reason about
- Concurrency is even more difficult to reason about
  - At the scale of datacenters and across datacenters
  - In the presence of failures
  - In terms of multiple interacting services
- Not to mention debugging...
- The reality:
  - Lots of one-off solutions, custom code
  - Write your own dedicated library, then program with it
  - Burden on the programmer to explicitly manage everything

What’s the point?

- It’s all about the right level of abstraction
  - Moving beyond the von Neumann architecture
  - We need better programming models
- Hide system-level details from the developers
  - No more race conditions, lock contention, etc.
- Separating the what from how
  - Developer specifies the computation that needs to be performed
  - Execution framework (“runtime”) handles actual execution

The datacenter is the computer!

The datacenter is the computer!
Big Ideas
Big Ideas

• Scale out, not up
  • Limits of SMP and large shared-memory machines
• Move processing to the data
  • Cluster have limited bandwidth
• Process data sequentially, avoid random access
  • Seeks are expensive, disk throughput is reasonable
• Seamless scalability
  • From the mythical man-month to the tradable machine-hour

Scale out, not up

- Prefer solutions that use many low-end machines instead of few high-end machines
- Typically, more than twice as expensive to double number of cores
- Commodity machines see better economies of scale
- What are reasons to scale up instead? Scale in?

Assume failures will happen

• Example: MTBF of 1,000 years (3 years)
• 1,000 node cluster will experience 1 failure per day
• System architectures must be designed with fault-tolerance in mind
  • Automatically take failed nodes offline
  • Resubmit failed jobs
  • Watch for stragglers

Good APIs hide system details

• Keeping track of details makes programming hard
• Counterexample: threading
  • Traditional means of parallelization
  • Many hazards: race conditions, deadlock, livelock
  • Difficult to reason about
  • Require higher-level design patterns (e.g. producer-consumer queues) to avoid pitfalls

Aim for ideal scalability

- N machines should handle (nearly) N times the load
- Avoid serial computation (dependencies)
- Avoid shared memory (side effects)
- Example: MapReduce

Move code to the data

• In recent years, CPUs have become much faster
• Storage has become much faster and more dense
• But network speeds haven’t changed much, so network is often the bottleneck for large-scale batch computation
• Solution: rather than fetching data from remote storage and processing it, move the processing code to the nodes that are storing your data

Avoid random disk access

• Random disk access is slow

• Example:
  • 1 TB database containing $10^{10}$ 100-byte records
  • Updating 1% of records will take one month on one machine
  • But rewriting entire database will take less than one day

• Solutions:
  • Process data sequentially
  • Don’t go to disk at all

Our Application:
Information Retrieval
Information Retrieval

• “Information retrieval (IR) is the activity of obtaining information resources relevant to an information need from a collection of information resources.”

• Very general because applications vary widely

• Typically:
  1. Index documents according to some model
  2. Use this index to find relevant documents
Documents

• Examples:
  • Web pages, emails, books, patents, images, time series

• Typically semi-structured
  • Some amount of structured metadata
    • E.g. URL domain
    • Easy to index and search
  • Main body is unstructured
    • E.g. web page text
    • Not so easy to index
Grep Is Not Enough

- In natural language there are many ways to express the same notion
- Some matches will be better than others
- Searching over every record isn’t going to scale
Documents vs. Records

- RDBMS records are typically highly structured
- RDBMS systems invariably use B-tree indexes for exact match record lookup
- For many IR applications, record-based data store is not a great fit
IR Concerns

- Relevance
  - Generally, whether the retrieved document meets the user’s information need
  - Retrieval models define relevance
  - Ranking algorithms use model to find relevant documents
  - Models typically capture statistical properties of the unstructured content
    - E.g. counting word occurrences vs. parsing sentences
IR Concerns

• Evaluation
  • How do we know if we’re doing a good job?
  • Test collections (TREC)
  • Metrics (precision, recall)
IR Concerns

• Meeting Information Needs
  • User interaction
  • Log mining
  • Query expansion, query suggestion
  • Relevance feedback
Search Engine Concerns

- Fast retrieval
- Fast, timely, and comprehensive indexing
- Scaling with users and data
- Application-specific issues (tuning, spam)
IR System: Bird’s Eye View
Index

Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham

1  2  3  4
blue   x   x   
cat    x   
egg    x   
fish   x   x   
green  x   
ham    x   
hat    x   
one   x   
red    x   
two    x   

Architecture circa 1999

Frontend Web Server

query

Index servers

\[ I_0 \quad I_1 \quad I_2 \quad \ldots \quad I_N \]

Doc servers

\[ D_0 \quad D_1 \quad \ldots \quad D_M \]

Index shards →

Doc shards ←

Source: Jeff Dean WSDM 2009
Architecture

Ad System → Frontend Web Server → Cache servers

Index servers: $I_0, I_1, I_2, \ldots, I_N$

Doc servers: $D_0, D_1, D_M$

Replicas: $I_0, I_1, I_2, \ldots, I_N$

Doc shards: $D_0, D_1, D_M$

Source: Jeff Dean WSDM 2009
Growth

Ad System → Frontend Web Server → Cache Servers

Index servers:

Index shards: $I_0, I_1, I_2, I_3, I_4, \ldots, I_{10}, \ldots, I_{60}$

Cache servers

Doc Servers

Source: Jeff Dean WSDM 2009
Circa 2004

Source: Jeff Dean WSDM 2009
Single-Threaded Asynchronous Execution
The Problem

• User makes a request to a server
• We need to spend a little time coming up with a response
• In the meantime, we still need to be able to accept new connections!
• What are some solutions to this problem?
Traditional Solutions

• Prefork processes
• Spawn a worker thread
• Disadvantages:
  • Thread and process overhead
  • Reasoning about multithreaded code
• Let’s talk about an alternative ...
Blocking vs. Non-blocking Sockets

• Blocking sockets: API calls will block until action (send, recv, connect, accept) has finished

• Non-blocking sockets: these calls will return immediately without doing anything

• In Python, use socket.setblocking(0) to make a socket non-blocking
Event Loop

• Single thread, single process
• Uses non-blocking I/O to wait for data to come back
  • Allows us to service new connections while we wait
• All non-I/O activity within the process will still block
Tornado

- Event loop-based web framework
- Started at FriendFeed
- Bought by Facebook
- Lots in common with Twisted
- Includes C10k web server
select vs. poll vs. epoll

• select – system call that allows a program to monitor multiple file descriptors (sockets)
• Builds a bitmap of all fds, turns on bits for fds of interest
• Each call is O(highest file descriptor)
select vs. poll vs. epoll

- poll – requires registering file descriptors of interest
- Each call is $O(\text{number of registered file descriptors})$
select vs. poll vs. epoll

• epoll – better event notification
• Same API as poll
• Each call is $O(\text{number of active file descriptors})$
import tornado.ioloop
import tornado.web

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")

if __name__ == "__main__":
    application = tornado.web.Application([
        (r"/", MainHandler),
    ])
    application.listen(8888)
    tornado.ioloop.IOLoop.instance().start()

class AsyncHandler(RequestHandler):
    @asynchronous
    def get(self):
        http_client = AsyncHTTPClient()
        http_client.fetch("http://example.com",
                        callback=self.on_fetch)

    def on_fetch(self, response):
        do_something_with_response(response)
        self.render("template.html")

Tornado Coroutines

class GenAsyncHandler(RequestHandler):
    @gen.coroutine
    def get(self):
        http_client = AsyncHTTPClient()
        response = yield http_client.fetch("http://example.com")
        do_something_with_response(response)
        self.render("template.html")

Until next time...

- Assignment 1 has been posted
  - Due next week
- My email: doherty@cs.nyu.edu
- Questions?