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** – Dynamo, Cassandra, Riak, Redis
- **Document** – MongoDB, CouchDB, Elasticsearch
- **Column** – VoltDB, Vertica
- **Graph** – Neo4j, OrientDB
- **BigTable** – HBase, HyperTable
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 – 7-9pm
• 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
• Due by start of class
Assignments

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

• **Resubmit policy**
  • Graded assignments may be resubmitted up to a week later for (up to) full credit
  • Must write detailed notes on what went wrong with your first solution
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
Big Ideas
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?

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

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
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 $O$ (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 before class next week

- Questions?