PART B. GEAR SESSIONS
SESSION 1: PETA-SCALE STORAGE SYSTEMS

Google had 2.5 million servers in 2016

Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs535

FAQs

• Quiz 1
  • Pseudocode should be interpretable as a MapReduce
    • Your code should be interpretable as a actual MR code
      • E.g.
        • Step 1. Read lines
        • Step 2. Tokenize it
        • Step 3. Group records based on the branch
        • Step 4. Sort all of the record of a branch
        • Step 5. Find the top 10 per branch
      • Can this code an effective mapreduce implementation?
    • <Key, Value> is the core data structure of communication in MR without any exception
• Next quiz: 2/21 ~ 2/23
  • Spark and Storm
FAQs

• How to lead the discussion as a presenter
  • GOAL: You should involve your audience to the discussion
    • Please remember that you have at least 10 other students (3 other teams!) who already read the same paper and submitted reviews!!
  • Initiate questions
    • “What do you think about this? Do you think that the approach XYZ is suitable for ABC?”
  • Provide discussion topics
    • “OK. We will discuss the performance aspect of this project. This project has proposed approach X, Y, and Z…”
  • Pose questions
    • “We came up with the following questions…”

Topics of Today’s Class

• Apache Storm vs. Heron
• GEAR Session I. Peta Scale Storage Systems
4. Real-time Streaming Computing Models: Apache Storm and Twitter Heron

Apache Storm

Apache Heron

Limitation of the Storm worker architecture

- **Multi-level scheduling** and complex interaction
  - Tasks are scheduled using JVM’s preemptive and priority-based scheduling algorithm
  - Each thread runs several tasks
  - Executor implements another scheduling algorithm

- **Hard to isolate its resource usage**
  - Tasks with different characteristics are scheduled in the same executor (e.g. Kafka spout, a bold writing output to a key-value store, and a bolt joining data can be in a single executor)
  - Logs from multiple tasks are written into a single file
  - Hard to debug and track the topology

![Diagram of JVM process and executors](image)
Limitation of the Storm worker architecture

- Limitation of the Storm Nimbus
- Scheduling, monitoring, and distributing JARs
- Topologies are untraceable
  - Nimbus does not support resource reservation and isolation
  - Storm workers that belong to different topologies running on the same machine
    - Interfere with each other
- Zookeeper manages heartbeats from workers and the supervisors
  - Becomes a bottleneck
- The Nimbus component is a single point of failure

Limitation of the Storm worker architecture

- If the receiver component is unable to handle incoming data/tuples
  - the sender simply drops tuples
- In extreme scenarios, this design causes the topology to not make any progress
  - While consuming all its resources
Apache Heron

- Maintains compatibility with the Storm API

- Data processing semantics
  - At most once – No tuple is processed more than once, although some tuples may be dropped, and thus may miss being analyzed by the topology
  - At least once – Each tuple is guaranteed to be processed at least once, although some tuples may be processed more than once, and may contribute to the result of the topology multiple times

Aurora Scheduler

- Aurora
  - Generic service scheduler runs on Mesos
Aurora Scheduler

- Each topology runs as an Aurora job
  - Consisting several containers
  - Topology master
  - Stream manager
  - Heron Instances
  - Generic service scheduler runs on Mesos

Topology Backpressure

- Dynamically adjust the rate at which data flows through the topology
  - Skewed data flows

- Strategy 1: TCP Backpressure
  - Using TCP windowing
  - TCP connection between HI and SM
  - E.g. for the slow HI, SM will notice that its send buffer is filling up
  - SM will propagate it to other SMs
Topology Backpressure

**Strategy 2: Spout Backpressure**
- SMs clamp down their local spouts to reduce the new data that is injected into the topology
- Step 1: Identifies local spouts reading data to the straggler HIs
- Step 2: Sends special message *(start backpressure)* to other SMs
- Step 3: Other SMs clamp down their local spouts
- Step 4: Once the straggler HI catches up → send a *stop backpressure* message to other SMs
- Step 5: Other SMs start consuming data

**Strategy 3: Stage-by-stage backpressure**
- Gradually propagates the backpressure stage-by-stage until it reaches the spouts
  - which represent the 1st stage in any topology

GEAR Session 1. Peta-scale Storage Systems
GEAR Session 1. Peta-scale Storage Systems

• Objectives
  • Understanding large scale storage systems and their applications

• Lecture 1. 3/17/2020
  • Distributed File Systems: Google File System I, II and HDFS

• Lecture 2. 3/19/2020
  • Distributed File Systems: Google File System I, II and Apache HDFS
  • Distributed NoSQL DB: Apache Cassandra DB

• Lecture 3. 3/24/2020
  • Distributed NoSQL DB: Apache Cassandra DB

• Workshop 3/26/2020

• Workshop 3/26/2020
  • [GS-1-A]
    • Presenters: Team 12 (Miller Ridgeway, William Pickard, and Timothy Garton)
  • [GS-1-B]
    • Presenters: Team 2 (Approv Pandey, Poomima Gunhalkar, Prinila Irene Ponnayya, and Saptashi Chatterjee)
  • [GS-1-C]
    • Presenters: Team 9 (Brandt Reutimann, Anthony Feudale, Austen Weaver, and Saloni Choudhary)
GEAR Session 1. peta-scale storage systems
Lecture 1. Google File System and Hadoop Distributed File System

This material is built based on

- Andrew Fikes, Storage Architecture and Challenges, Faculty Summit, 2010
- Jeff Dean's SOCC keynote, Building Large-Scale Internet Services
- http://sysmagazine.com/posts/206986/
- Erasure Coding: Backblaze Open sources Reed-Solomon
  - https://www.backblaze.com/blog/reed-solomon/
- An introduction to Reed-Solomon codes
The Machinery

- Servers
  - CPUs
  - DRAMS
  - Disks

- Racks
  - 40-80 servers
  - Ethernet switch

- Cluster
  >10,000 nodes

Google Cluster Software Environment

- Clusters contain 1000s of machines, typically one or handful of configurations
  - File system (GFS or Colossus) + cluster scheduling system are core services

- Typically 100s to 1000s of active jobs
  - mix of batch and low-latency, user-facing production jobs
The Realistic View of a Data Center

- Typical first year for a new cluster:
  - ~1 network rewiring (rolling downtimes: ~5% of machines over 2-day span)
  - ~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
  - ~5 racks go wonky (40-80 machines see 50% packet loss)
  - ~8 network maintenances (4 might cause ~30-minute random connectivity losses)
  - ~12 router reloads (have to immediately pull traffic for a couple minutes)
  - ~3 router failures (have to immediately pull traffic for an hour)
  - ~dozens of minor 30-second blips for DNS
  - ~1000 individual machine failures
  - ~thousands of hard drive failures
    - slow disks, bad memory, misconfigured machines, flaky machines, etc.
  - Long distance links
    - Reliability/availability must come from software

Numbers we should know

- Level 1 cache reference
  - 0.5 ns
- Branch misprediction
  - 5 ns
- Level 2 cache reference
  - 7 ns
- Mutex lock/unlock
  - 25 ns
- Main memory reference
  - 100 ns
- Compress 1KB with cheap compression algorithm
  - 3,000 ns
### Numbers we should know

- Read 1 MB sequentially from memory
  - 250,000 ns
- Round trip within the same datacenter
  - 500,000 ns
- Disk seek
  - 10,000,000 ns
- Read 1 MB sequentially from disk
  - 20,000,000 ns
- Send packet CA->Netherlands->CA
  - 150,000,000 ns

---

### Back of the Envelope Calculation

- How long to generate an image results page (30 thumbnails)?

  - **Design 1:** Read serially, thumbnail images (256KB) on the fly
    - 30 seeks * 10 ms/seek + 30 * 256K / 30 MB/s = 560 ms

  - **Design 2:** Issue reads in parallel:
    - 10 ms/seek + 256K read / 30 MB/s = 18 ms

- Lots of variations:
  - caching (single images? whole sets of thumbnails?)
  - pre-computing thumbnails
  - ...
Storage Software: GFS

• Google’s first cluster-level file system (2003)
  • Designed for batch applications with large files. Single master for metadata and chunk management.Chunks are typically replicated 3x for reliability.

• Lessons
  • Scaled to approximately 50M files, and 10PB
  • Large files increased application complexity
  • Not appropriate for latency sensitive applications
  • Scaling limits added management overhead

Storage Software: Colossus (GFS2)

• Next-generation cluster-level file system
• Automatically sharded metadata layer
  • Data typically written using Reed-Solomon (1.5x)
  • Client-driven replication, encoding and replication
  • Metadata space has enabled availability

• Why Reed-Solomon?
  • Cost. Especially with cross cluster replication
  • More flexible cost vs. availability choice
Storage Landscape

- Early Google:
  - US-centric traffic
  - Batch, latency-insensitive indexing processes
  - Document "snippets" serving (single seek)

- Current day:
  - World-wide traffic
  - Continuous crawl and indexing processes (Caffeine)
  - Seek-heavy, latency-sensitive apps (Gmail)
  - Person-to-person, person-to-group sharing (Docs)

Storage Landscape: Flash (SSDs)

- Important future directions:
  - More workloads that are increasingly seek heavy
  - 50-150x less expensive than disk per random read
  - Best usage is still being explored

- Concerns:
  - Availability of devices
  - 17-32x more expensive per GB than disk
  - Endurance not yet proven in the field
GEAR Session 1. peta-scale storage systems
Lecture 1. Google File System and Hadoop Distributed File System

1. Google File System

Demand pulls in GFS (1/2)

- Files are huge by traditional standards
- File mutations predominantly through *appends*
  - Not overwrites
- Component failures are the *norm*
- Applications and File system API designed in *lock-step*
Demand pulls in GFS (2/2)

- Hundreds of producers will **concurrently append** to a file
  - Many-way merging

- High **sustained bandwidth** is more important
  - than low latency

The file system interface

- Does not implement standard APIs such as POSIX
- Supports create, delete, open, close, read and write

  **snapshot**
  - Create a fast copy of file and directory tree

  **record append**
  - Multiple files can concurrently append records to the same file
    - Without additional locking
Architecture of GFS

- Client
- GFS Master
- Client
- GFS Chunk Server
- Linux File System
- GFS Chunk Server
- Linux File System
- GFS Chunk Server
- Linux File System

Chunks

- Obvious reason
  - The file is too big

- Set the stage for computations that operate on this data
  - Parallel I/O
  - I/O seek times are $14 \times 10^6$ slower than CPU access times
### Chunk size

- This is fixed at **64 MB (→Now 128MB)**
  - Much larger than typical FS block sizes (512 bytes)

- **Lazy space allocation (delayed space allocation)**
  - Stored as plain Linux file
  - Physical allocation of disk space is delayed as long as possible
    - Until data at the size of the chunk size
  - Extended only as needed
  - Avoiding internal fragmentation

### Chunk size: But why this big?-Advantage

- **Reduces client interaction** with the master
  - Can cache info for a multi-TB working set

- **Reduce network overhead**
  - With a large chunk, client performs more operations
  - Persistent connections

- **Reduce size of metadata** stored in the master
  - 64 bytes of metadata per 64 MB chunk
Large chunk size: Disadvantage

- Small files (with small number of chunks)
  - May become hot spots
  - e.g. popular executable files

- Solution
  - Assigning a higher replication factor

GEAR Session 1. peta-scale storage systems
Lecture 1. Google File System and Hadoop Distributed File System
2. Master Operations
Architecture of GFS

- Client
- GFS Master
- GFS Chunk Server
  - Linux File System
- GFS Chunk Server
  - Linux File System
- GFS Chunk Server
  - Linux File System

Master operations

- Single master
- Manage system metadata
- Leasing of chunks
- Garbage collection of orphaned chunks
- Chunk migrations
**ALL system metadata is managed by the Master and stored in Main Memory**

- File and chunk namespaces
- Mapping from files to chunks
- Location of chunks

**Logs mutations into a permanent log**

---

**Size of the file system with 1 TB of RAM: Assume file sizes are exact multiples of chunk sizes**

- Assume that the chunk size is 64MB ($2^6 \times 2^{20}$)
- The file namespace data: less than 64 bytes
- Number of entries = 1TB/(size of namespace data) = $2^{40}/2^6$

**MAXIMUM SIZE of the file system**

= Number of entries x Chunk size

= $\frac{2^{40}}{2^6} \times (2^6 \times 2^{20})$

= $2^{60} = 1$ EB
Tracking the chunk servers

- Master **does not** keep a persistent copy of the location of chunk servers
- List maintained via **heart-beats**
  - Allows list to be in *sync* with reality despite failures
  - Chunk server has final word on chunks it holds

---

Simple **read** example

![Diagram of read example](http://www.cs.colostate.edu/~cs535)
Questions?