Reading - MapReduce paper

[[MapReduce - Simplified Data Processing on Large Clusters.pdf]] Responsibilities: - User: mapper and reducer - Run-time: partitioning alg, parallelizing/scheduling across machines/cores, load balancing, IPC, and failure handling (re-execution) - Easily scalable to process huge data (crawled, web request logs, etc.) - Avoid obscuring relatively straightforward computations for necessarily scaling computations to big data Programming Model: - Map - Input: key/value pair - Output: set of intermediate key/value pairs - Reduce - Input: 1 key —> [value] - Might be iterative to save memory - Output: 1 key —> 0-1 values - Marshalling: converts to strings in C++ library when sending btwn workers Parallel: - map task split into M pieces - Reduce task split into N: - Hash keys via hash(key) % N - master assigns work to workers - Map worker reads its split and parses —> sends to Map func —> buffer output to RAM - Periodically write to that worker's LOCAL disk (corresponding N splits) and notify master of locations - Master assigns reduce work to worker - Reduce worker RPCs map worker for their buffer intermediate key-value pairs - Reducer sorts and iterates through each key, passing into reduce func —> output to its own reducer file on GLOBAL disk - Output of R files, can use another mapreduce to combine Fault Tolerance - Master keeps track of state of tasks (in progress, ready, done) and output locations - Periodically pings all in-progress workers, if no response, marks task as ready for next worker - Bc intermediate work stored on failed machine's local disk, inaccessible - Then notifies all reduce workers of this change so they read from correct worker File system organization - one map worker done: writes to R temporary private files on local disk, provides the R names to master - One reduce worker done: writes to one temporary output file, renames to final output (atomic rename) - Prefer to schedule reducer to workers already having its partition or close to (i.e. same network switch) one to conserve bandwidth Parameters - M,R ideally big to easily load balance and recover from faults - Master node complexity: O(M+R) for scheduling, O(M*R) for tracking state - Keep R reasonably smaller than M bc of # output files Stragglers - when master close to done (99%), spawns backups for all remaining in-progress workers Refinements - custom partition algs: i.e. group together hostname urls: hash(Hostname(urlkey)) mod R - Guaranteed sorted key output order - Combiner: "the1" "the2" "the3" to reduce workload - Does partial merging in mapping workers - Same as reducer but not write to output file, rather intermediate - Custom input type: define simple reader that reads records from file format or remote database - Custom extra output files for mappers/reducers (auxiliary, require atomic writes/renames, use potent, deterministic) - Skipping deterministic error on particular records: seqno sent by workers, master sees record fail more than once, tells worker to skip it - Sequential version for debugging purposes (gen) and masking out specific map tasks - Localhost browser view of map task status (in progress, stderr, stdout, failed, etc.) - Counter: to keep track of some statistic of entire mapreduce task - Counter in each map and each reduce worker - Periodically piggybacks on master pinging workers to be aggregated/displayed in master status page - Avoids dupe counting when firing backup workers - Grep experiment - Lots of M splits into one R output (bc searched grep is relatively rare in records) - Overhead of propagating code to the workers still quite fast - Sort experiment - Convert record into sort key —> original record, hash into R outputs based on first few bytes of sort key - Assumes built-in knowledge of key distribution for partitioning Use cases

large-scale machine learning problems, • clustering problems for the Google News and Froogle products, • extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist), • extractionofpropertiesofwebpagesfornewexper- iments and products (e.g. extraction of geographi- cal locations from a large corpus of web pages for localized search), and • large-scale graph computation

Benefits - easy for non-distributed programmers thanks to restrictive design - Good performance to avoid combining unrelated code for the sake of "less passes" (modularity) - Performance easy to operate bc all failures handled automatically # Lecture

hard to debug multiple errors network error can check via checksum, but cpu computation error cannot detect insidious - spread over entire system if not stoped transient? race conditions? time-based non-deterministic

final exam tip: best to avoid distrib. system unless very needed, might be single app sys

Eight Fallacies of Distrib. Systems

1. network is reliable (leaky abstractions, TCP)
2. latency is 0 (even if super fast within data center)
3. bandwidth is infinite (assuming easily propogate data within nodes)
4. network secure
5. topology not change (no elasticity principle or scalability)
6. there's one admin (diff admins, diff goals, etc.), or malicious admins/actors
7. transport cost is 0 (cloud)
8. network homogenous (old nodes, etc.)

Example: - tried to scrape all of Facebook BFS - **separation of concerns**: all in one python file: all stored in ram, scrape ppl & parse at same time - lots of data loss, bugs from unhandled exceptions, lost work queue (in RAM) - facebook return blocking/corrupted data - transient mem - **being rate limited via IP address, from pausing** 1. separate into scraper to disk + extracting friend list (processing) 2. work queue (Redis) in RAM, retry if failed add back to queue until TTL, and add recovery incase node failed 3. **Middleware:** separate sysetm that runs across multiple nodes, i.e. cloud to store all files in one place - in buckets i.e. AWS like folders but not nested

Thought Experiments

General Problem; **Byzantine Generals**: 100% reliable messengers but possibly traitors (mislead other general) - malicious generals can work together or be controlled by one dictator - easier to solve w/ cryptography to prove what was said

:= declaration and asignment (foo := 32 or var foo int foo=32) packages, imports, main func git bisect: binary search to find erronious commit go run test.go

package main import "fmt" func main() { fmt.Printf("Hello world") }

Reading - Kleppmann Ch 10, MapReduce Section

> [!PDF|] [[Martin Kleppmann - Designing Data-Intensive Applications The Big Ideas Behind Reliable, Scalable, and Maintainable Systems (2017).pdf#page=419&selection=61,0,61,37|p.397]] > MapReduce and Distributed Filesystems

Mapreduce section

- similar to ONE unix log processing
  - blunt, effective, simple
  - Read set of input and break into records
  - self-contained, no side-effect, read-only
  - but writes to distrib. file sys (i.e. Hadoop HDFS, GlusterFS, GFS, etc.)
    - HDFS "shared-nothing": any computer hardware
    - daemon process per machine w/ network api
    - central server NameNode tracks/replicates into 1 big filesys (like RAID)
- ONE mapreduce job execution
  1. read input files into small records
  2. Call mapper to extract key+value from each record
  3. sort key-value pairs by key
  4. call reducer to iterate over sorted and combine (adjacent after sorting)
- CONCEPTUAL MAP/REDUCE MNEUMONICS:
  - Mapper goal: put data in easy form for sorting
  - Reducer goal: summarize/merge the sorted dupe keys
  - Mapper: sends msgs to reducers, key = destination address of reducer
- Vs Unix pipeline:
  - can "pipe" first mapreduce job directly to second ("workflows")

- BUT like list of cmds writing to temp file vs pipe buffers
- BUT distributed across all machines implicitly vs unix
- PARTITION: 1 job input = directory
  - each file in directory = 1 partition = 1 mapper (M is fixed)
  - vs arbitrarily chosen R reducers by author, (key mod R)
  - Scheduler starts mapper on each machine w/ ***replica*** of file (passing records to mapper)
  - "Putting computation near the data" - locality
  - Copies code to appropriate machine before starting
- Reducer: sort too big! instead in STAGES
  - mapper partitions via % R, outputs sorted partitions
  - reducer merges sorted partitions so adjacent regardless of diff/same mappers
- indexing vs full table-scans
  - indexing: quickly easily load info on specific user/a few records
  - table scan: calculate aggregates over lots of records, join 2 tables (resolve all occurrences)
  - example of JOIN: user id –> user profile![[Pasted image 20241031221436.png]]
    - goal: get age demographic of urls
    - indexing: $$$$$$$: iterate user activity, query id,
      - round trip overhead, locality out of your hands, overload database?
      - nondeterministic, race condition if remote database changes
    - table-scan (**sort-merge join**):
      - locality & deterministic: *copy* user database near user activity log
      - map reduce to bring together both files efficiently: here M=2 (activity log + database) and R = 2 (even and odd ids)
      - ![[Pasted image 20241031222629.png]]
      - auto sorts even/odd partitions so that records from **both** files with same IDs adjacent.
      - reducer: iterates even ids, calls reduce, outputs viewed-url, viewer-age pairs
        - no need for network or huge memory
  - GROUP BY (group by key then sum/count/top k/aggregate)
    - have mapper's key-value pairs correspond to desired grouping
    - sessionization: get all activity events of particular user session from various server log files via cookie/id/etc.
- skew/linchpin dealing
  - **reduce-side joins** - join logic in reducers
    - pro: agnostic to input as mapper is same
    - con: sort, merge, replicate to r reducers per hot key $$$$ mem writes
    - Pig skewed join (parallelized randomization)
      1. samples job to find hot keys (specific user A tons of activity in log)
      2. sends all activity records of user A to random subset of r reducers (nondeterministic) instead of one reducer (reducer no. A mod R)
      3. each of the r out of R reducers processing user A also needs user A's database entry (the other input) so replicated across all r.
      - must replicate other file (user A's database entry) to all reducers $$$$$$
    - Crunch sharded join
      - same as Pig but explicitly define the hot keys beforehand
  - **map-side joins** - faster at cost of assumptions to input
    - no reducers or sorting. each mapper reads 1 input file block, writes 1 output.
    - **broadcast hash joins** (all in local mapper mem)
      - assumes: small & large dataset, small one fits into mappers' own mem
      - i.e. user database –> in-mem hash table in each mapper –> lookup id per user activity log
      - each mapper tasked with 1 file block of larger dataset (i.e. user activity log) that can be loaded entirely to mem
      - solution if can't fit in mem: load smaller database in read-only index on local disk, auto cached by OS RAM replicates same behavior
    - **partitioned hash joins**
      - partition both inputs (user database, user activity log) w/ same algorithm (i.e. even/odd)
      -
    - Hive skewed join
      - like Crunch, need to explicitly define hot keys within table metadata

- stores all hot key records separately from rest
- then runs map-side join