CS262, Fall 2008 : Parallel DBs and MapReduce

- **Parallel Databases**
  - History of Database Machines vs. Commodity HW.
  - Side benefit of relational model: parallelism.
- Basic concepts
  - Pipeline vs. partition parallelism
  - Speedup (fixed problem size) vs. Scaleup (problem and HW grow)
  - Barriers to parallelism: Startup, Interference and Skew
    - Note in paper about interference: 1% slowdown limits scaleup to 37x
  - Shared mem & disk revisited
    - Inevitably you want caching, leading to processor affinity. Why not code it up?
- DB dataflow models: iterators and pipelines. more on this next time!
  - Hashjoin and sort algorithms.
- Sort benchmarks, balancing the HW pipeline
  - Ramification for manycore? datacenters?
  - The "hard stuff": DB layout, query optimization, mixed workloads, UTILITIES!
- **MapReduce**
  - Goals
    - automatic parallelization/distribution
    - fault-tolerance
    - I/O scheduling
    - status/monitoring
  - Structure
    - Pasted Graphic
    - map (k1, v1) -> list(k2, v2)
    - reduce(k2, list(v2)) -> list(whatever)
- Platform:
  - Commodity PCs (dual processor), commodity NW, 100's/1000's of machines, cheap IDE disks
  - GFS for reliable file storage
  - job = {tasks}, passed to scheduler
- Basic Execution:
  - data "automatically" partitioning into M "splits". Reduce done by hashing mod R. M and R specified by the user.
  - Splits are of a single "file" into physical chunks (# of Bytes)
  - Mappers write to memory buffers. Periodically, buffers are flushed locally. Location sent to master.
  - Master notifies Reduce workers about flushed results of Mappers. Fetches them via RPC.
  - Reduce then performs Sort-based groupBy. Output appended to final output file in GFS for this reduce partition.
  - When all M's and R's done, Master wakes up user program to "return" from the MapReduce call. R Reduce files are left in place, possibly to be reused in another MapReduce stage.
- FT:

CS262, Fall 2008: Parallel DBs and MapReduce

- Master pings workers for failure.
- Completed map tasks are reset as "idle" (i.e. not yet started) because they're inaccessible on the failed node's disk. In-Progress Maps and Reduces also set as "idle". Completed Reduces are safely in GFS.
- All reducers need to be told of a failure, due to "pull" model.
- Master failure can be handled by checkpointing of worker state.
- To ensure correctness, need ATOMIC commit of map and reduce. So tentatively write to private temp files (R for M's, 1 for R's).
- When Mapper completes, it sends message to master with the names of the R files, which it records in its data structure. Subsequent such messages ignored.
- When Reduce completes, worker renames temp output to final output name. Atomic rename in GFS ensures that only one of possibly many redundant reducers "wins".
- Clearly, non-deterministic functions provide loose semantics.

- Locality:
  - Master assigns Mappers with an understanding of where the GFS blocks of the input are. Best on a machine with a replica, else "close" in the network (same switch).
- Granularity:
  - Many more M's and R's than machines. Good for load-balancing, and you need to LB after failures. Since master has to bookkeep the M's and R's -O(M*R) state and O(M+R) scheduling decisions, don't want this insanely big. Also, may not want too many R's cos result is spread into many files.
  - Rule of thumb: choose M to be about 16MB-64MB, R a small multiple (e.g. 100) of the #machines. 2K machines, M=200K, R=5K.
- Stragglers:
  - Note causes: faulty though correctable disks, shared resource utilization, bugs in code.
  - One solution: competition. Redundant execution of the last in-progress tasks. When useful, when not?
- Combiner function: for commutative/associative Reduce
  - partial pre-aggregation at the mapper. output to the local intermediate file to be sent to reducer.
- Other stuff
  - Side effects: up to the programmer to make atomic and idempotent.
  - Optionally skip bad input records: signal handler sends a UDP packet with seqno to Master. If it sees >1 such, skips it next time.
  - Sequential local version for debugging.
  - HTTP server in master for status, stderr, stdout
  - Running aggregation of "counters" for sanity check
- Performance
  - 1800 machines, 2GHz Xeons, 4GB RAM< 160GB disk (!!), Gb Ethernet. Two level tree switched net with 100-200 Gbps aggregate at root
  - Grep experiment: see startup cost in graph -1 minute startup, 150 seconds total! Startup includes code propagation, opening 1000 files in GFS, getting GFS metadata for locality optimization. WOW.

CS262, Fall 2008: Parallel DBs and MapReduce

  - Sort: 891 seconds, 1800 machines, 3600 disks. 1057 seconds was TeraSort benchmark. (2006 was 435 Secs on **80 Itanium machines** with 2520 disks, 2000 was 1952 IBM SP machines with 2168 disks!)
- Note FLuX project (ICDE 03, SIGMOD 04)
  - Fully pipelined
  - Process pairs + Partitioning
  - quite complex, but more powerful
- Big picture questions
  - when is complexity merited? Architectural elegance in MapReduce?
  - Programming models