CPSC 436: Cloud Computing

Data Processing

- for large-scale data processing, we want to distribute computation, easy-to-use API, fault tolerant

MapReduce

- shared nothing architecture (pass info through network)
- provides: data distribution, fault tolerance, load balancing
- split the data up, process data partitions (map), group the partitions results (shuffle), aggregate them back together (reduce)
- node (machine) types: master & worker (map + reduce workers)
  - master: assigns work to workers
  - map workers: read content, run the map function, results are written to disk
  - reduce workers: read the intermediate results, run reduce func
- fault tolerance: basically re-run the task on another node
  - special case: reduce task finished - this would have been saved to global HDFS (so replicated already) so we'll have it
  - it also saves the state of the master node periodically
- map reduce fails for complex functions

Apache Spark

- MR share data by storing them to HDFS global storage (slow), Spark uses memory instead
- big idea: read input data once and process multiple query in mem
- architecture: driver process & executor process
  - driver: runs the main function, assigns jobs to executors, optimize and distribute work across executors
  - executors: execute code assigned by driver, report progress
- Resilient Distributed Dataset (RDD):
  - RDD is a distributed, and fault tolerant data type used in Spark (can partition to different nodes, is replicated)
  - **it is immutable** (so output is actually a new RDD)
- RDD operations: transformations & actions
  - transformations: take on RDD and produces another RDD; transformations are lazy, creates a flow-chart/logical plan
  - actions: RDD operations that produce non-RDD output; they trigger the transformations required on the RDD
  - note: "collect" is an action bc it returns a local (non RDD) list
- caching & checkpointing: to save data for later jobs to use → cache saves RDD to memory, checkpoint saves RDD to disk
- workflow
  - lineage creation: create DAG from list of transformations → use to recompute lost data in case of failure
  - stage identification: Spark optimizes the DAG, splits up the operations that can be done in parallel into **stages**
  - job scheduling: Spark's scheduler launches task for each stage (likely to launch task on node that has the needed data)
  - shuffling: redistribute data to nodes between stages (likely place for bottlenecks)
  - task execution: tasks within a stage are executed
- fault tolerance: **not as fault tolerance as MR**, uses the lineage graph to reproduce lost data, can also checkpoint b/t stages

Structured Data Processing (SparkSQL)

- Spark + RDD is unstructured (no columns) + users like writing declarative queries (specify what, not how; like in SQL)
- SparkSQL: module in Apache Spark that allows you to execute SQL-like queries on structured data
  - uses DataFrame & DataSet instead of RDD (have columns)
- DataFrames (DF): distributed, and fault tolerant data type with named columns (similar to tables in relational database)
  - built on top of RDD; allow for SQL-like operations
  - columns have names and types so you can index via String
  - rows have to be indexed by their row number
  - DF transformations: DF in → DF out, **still lazy**
  - DF actions: DF in → non-DF out
  - aggregation: run some agg function over key/grouping
  - temporary view: store a DF as temp table in memory
- DataSet (DS): distributed, and fault tolerant data type where row is typed (i.e each row is a instance of a Person class with attributes)
  - basically strongly-typed version of DF → provides type safety
  - transformations (lazy), actions, aggregation are all same as DF
- structure data execution:
  - write DF/DS/SQL code
  - if valid code - Spark turns it into a logical plan (DAG, lazy)
  - Spark optimize this logical plan and turns it into physical plan (specify how logical plan executes on the cluster - takes into account network, CPU, memory, etc)
  - Spark executes physical plan on the cluster, return result to user

Distributed Machine Learning

- aside: Spark ML
  - MLLib is a package built on Spark - interface for doing ML
  - uses RDD or DF so its fault tolerant and distributed
  - pipeline: sequence of algo to process + learn from data;
  - transformers: DF in → DF out (model itself is a transformer)
  - estimator: DF in → non-DF out (usually output is model itself)
- motivation: model & data size has increased so much, want to spread out the load → **model parallelism** or **data parallelism**
  - model parallelism is out of scope → split up the model

Data Parallelism

- data parallelism: partition dataset and process partitions on different nodes in parallel; after processing individual gradients are averaged and updated
  - need to consider the following design choices: aggregation algorithm, communication frequency, data compression
- aggregation algorithm: how to agg grad - consider following 3
  - Parameter Server (centralized): use a central server to store global model param; workers compute locally then send to PS; PS aggregate them and improve global params
  - AllReduce (decentralized): params are stored on all workers; workers exchange their local results via AllReduce function (broadcast it to everyone, grab from everyone) → there's always a consistent state across the network
  - Gossip: no PS; workers give local results to immediate neighbors; consistency achieved at the end of algorithm
- communication frequency: how often you exchange info; more frequent means more network overhead
  - Synchronous: workers sync after every iteration; bottlenecked by slowest node (stragglers)
  - Stale-Synchronous: tolerates some set amount of delays
  - Asynchronous: workers send result when done; no waiting
  - Local SGD: worker does several iterations b4 synchronizing
- data compression: help network traffic overhead
  - quantization: represent data using fewer bits (i.e rounding)
  - sparsification: send less elements (i.e only send non-zero grad)

Stream Processing
- definition: continuously taking in data and computing new results
- input data is unbounded (i.e posts on SM) → we want to process the data as it flows (without storing persistently)
- need to use Data Streaming Management System (DSMS)

Data Stream Storage
- need to be able to take streams of unbounded data, process them, then send result to consumers (i.e applications)
- need a messaging system → notify user as there are new data
- direct messaging: cons & prod connected to each other → result pushed directly to consumer whenever data is created (socket)
  - both parties need to be online at the same time (can be bad)
  - good for latency critical applications
  - if produce rate > consume rate → messages are dropped
- message broker: broker runs as server, producers and consumers connects to it as clients
  - producers write messages to the broker, and consumers receive them by reading from the broker (usually does this async)
- partitioned log: logs are messages that are stored for a specified amt of time - allows consumers to consume later
  - log is append only; can be partitioned and stored on diff nodes

Kafka (Log-based Message Broker - Stream Storage)
- it's distributed, topic-oriented, partitioned, replicated
- distributed: there are many broker servers working together (appear as 1 to clients)
- topic-oriented: data are organized into topic (categories)
- partitioned: each topic split into multiple partitions (allows for parallelism R/W)
- replicated: each partition replicated across nodes for fault-tol
- brokers in Kafka are stateless (consumer maintains their offset)
- Kafka only guarantees at-least-once delivery; also guarantee that messages within a single partition is delivered in order
- has zookeeper that's is the coordinator and orchestrator

Data Stream Processing
- tuple: a record of data; can have many fields; processed by PE
- processing elements (PE): unit of computation in streaming app
  - take input tuple, apply a function to them, produce output tuple
  - note: PE can be stateful or stateless
- job: essentially a unit of work executed by processing system
  - job consists of (parallelizable) tasks that need to be executed
  - tasks within job are executed by PE (do the actual processing)
- job management: tracking and managing jobs w/ their PEs
  - must identify and track PEs, the jobs they belong to, and the user who instantiated them
- parallelism: essential for scaling applications
  - pipelined: execute different computation stages (there's ordering involved) concurrently (like y86 in 313)
  - task: execute independent tasks of computation concurrently
  - data: execute same computation in parallel on diff data partition
- parallelism challenges for stream processing
  - must handle bursty loads → scale horizontally
  - scaling horizontally complicates maintaining fault-tolerance → managing the state and coordination between these distributed components becomes more complex (states might also be lost when there are failures)
- point: parallelism + maintaining states + fault tolerance is hard → use Spark Streaming

Spark Streaming (Data Stream Processing App)
- uses Micro-batch processing and is a Declarative API
  - micro-batch: break down the stream of incoming data into small chunks and process the chunks individually
  - declarative API: again, specify what but not how (i.e Python)
- main idea: treat streaming computation as a series of very small, deterministic batch jobs
  - chops the live data stream into batches of X seconds
  - each batch is treated as RDD (distributed + fault-tolerant)
  - results of RDD operations are returned in batches → called Discretized Stream (DStream)
- DStream: sequence of RDDs representing stream of data (each RDD in DStream contains data from certain interval of time)
- transformations: DStream in → DStream out
  - uses the same transformations we've seen → **still lazy**; actual computation initiated by a start() method

Graph Processing
- graph = relationships → lots of problems can be modeled as graphs and solved via graph algorithms
- graph algorithm challenges
  - difficult to partition data: graph nodes are connected and thus requires a lot of communication between clusters
  - difficult to parallelize computation: most graph algorithms are inherently sequential and resits parallelism
- graph partitioning: partition large graphs to distribute to hosts
  - try to create clusters based on traffic → want to limit inter-cluster communication
  - 2 main approaches: edge cut vs vertex cut
- edge cut: cut along edges to split graph into disjoint clusters
  - has high communication overhead (because edge spans multiple partitions) → bad for load balancing
  - good for node-centric communication where inter-node connectivity is rare
- vertex cut: divide edges of graphs into disjoint clusters (vertices w/ edges in different partitions are replicated across partitions)
  - good for edge-centric communication → high degree = good
- can't use Spark for graphs natively → graph processing uses iterative algos; don't align well with Spark's batch-oriented model

<u>Graph Parallel</u>
- each vertex compute individually (in parallel) → think like a graph
- computation typically depends on neighbors; after performing local comp, vertex sends message to neighbors and info propagate
- process synchronized by supersteps → in each superstep, vertices perform their comp and send messages concurrently
- problem with graph parallel computation: graph-parallel systems may impose restrictions to optimize graph processing performance
  - most of the time, your entire pipeline isn't all graph algo
  - ex. ETL & Analyze step might do better in data-parallel system

<u>Graph X</u>
- library to perform graph processing in Spark → think like a table
- it creates a table view from a graph data
  - ex. each view has its own operator and its own benefits
- support variety of graph algos to measure and analyze relationships
- provides powerful framework for running graph algo at scale, built on top of Spark's core features
- uses RDD underneath so distributed and fault-tolerant

Resources Management

- motivation: diff applications require diff frameworks; but running each framework on its own cluster is wasteful (hard to share data)
- solution: run various frameworks within same cluster → improves utilization and easier to share data; but need way to split resources

Mesos

- architecture
  - master: manages cluster resources → decide how to allocate resources; make resources offers based on demand and supply
  - framework scheduler: decides to accept or decline an offer; if accepted then scheduler is responsible for assigning tasks to resource provides → **each framework has a scheduler**
  - agents (aka slaves): run on node & actually executes the tasks; also report resource availability to master → 1 agent/node
- computation model
  - resources offers: resources (i.e CPU, memory) are offered to framework (Hadoop, Spark, etc) by the master
  - framework scheduler can accept or decline offer based on need
  - if framework accepts; it launches tasks to run on Mesos agents
- fine-grained sharing: resources can be be divided across multiple frameworks in small portions → efficient utilization
- system can dynamically allocate more or less resources based on needs of certain frameworks at certain times
  - Mesos assign resources based on the DRF algorithm
- fault tolerance and scalability
  - multiple master for high-avail → standby master can take over
  - if agent fail, master can detect and re-allocate tasks that were on failed agent to another live agent
  - Mesos let frameworks to checkpoint their state for recovery

Yarn

- uses containers → each application runs on an isolated container
- architect: resource manager, application master and node manager
- resources manager: central authority managing resources across the cluster
  - actually allocate resources to applications
- application master: unique to each application
  - negotiates resources with the resource manager
  - monitors task progress; manages task execution & recovery
- node manager: tracks and reports a node's resource usage
  - manages and monitors containers; enforce security

Cloud Security

- data is the main asset in the cloud → it's unique to the org (can't be replaced, re-bought or rebuilt); need to be protected
- **cloud security is hard** because it extends the attack surface (multi-tenancy + a lot of network communication)
- multi-tenancy: multiple independent users share the same physical infra; with possibly opposing goals
  - fix: inc isolation (VPC) & increase trust (SLAs) for tenants
- loss of control: data stored by cloud provider = potential loss of control over data → users rely on the cloud provider for data security, privacy, and resource availability
  - fix: use modern identity service (i.e OAuth + OpenID Connect)
- aside: User IDentification Federation allows user's single identity to be used across diff domains and cloud services/provider
- 3 entity-representation in the cloud: user, cloud, service
  - think of it as bidirectional graph between all entities
- steps to attack the cloud (not easy; but doable)
  - finding victim's location: IP address geo-location, network tracing & latency
  - being victim's co-resident: choosing same region, brute force placement, leveraging placement locality
  - exploiting co-residence: cross-VM attacks for info leakage, side channel attacks (learn about physical implementation)

OAuth + OpenID Connect

- OAuth solves an authorization problem (who has access to what) OpenID Connect solves an authentication problem (verify identity)
- **OAuth**: let users give third-party access to their info w/o password
- how OAuth works
  - user wants to let a third-party app (Client) to get their photos from a pic-hosting site (Resource Server) w/o sharing password
  - user redirected Authorization Server (i.e Google Photos)
  - user logs in - IdP asks the user for permission to share specific data or access rights with the third-party app (scopes)
  - after consent, the identity provider generates an authorization code and send user back to the original site
  - application sends the authorization code to the IdP token endpoint, get back access token & refresh token
  - can use this access token to access resources
- **OpenID**: used for SSO, let users log in once across multiple sites
- how OpenID works - relies on an Identity Provider (IdP - ex. FB)
  - ex. user wants to log in to a new online forum (Relying Party) using their existing Google account
  - Relying Party redirects the user to the OpenID Connect IdP (e.g., Google) for authentication
  - they log in using their IdP credentials; on success, IdP creates an identity assertion or token to confirm user's identity
  - user redirected back to the original site with ID assertion
  - Relying Party verifies assertion using IdP's public key
  - if assertion is valid, site logs the user in, granting them access