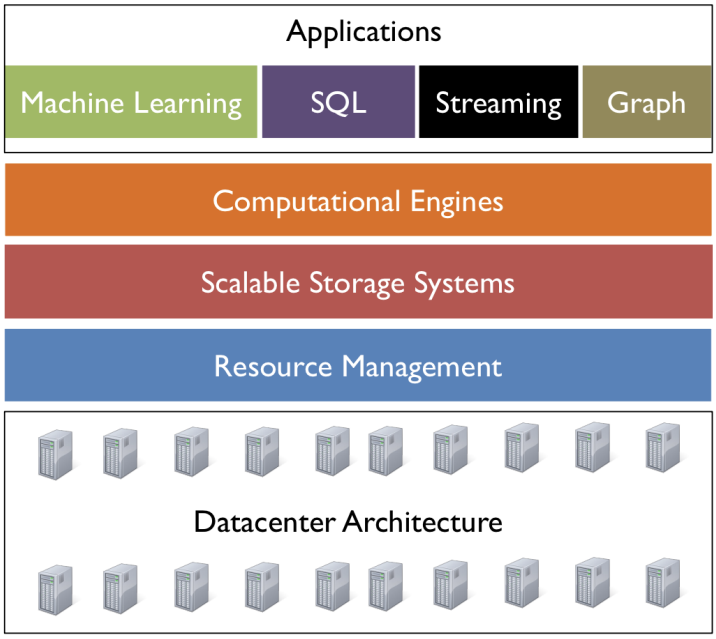
Big Data Systems

> Author: Guanzhou (Jose) Hu 胡冠洲 @ UW-Madison CS744
>
> Teacher: Prof. Shivaram



Infrastructure

Datacenter cluster architecture, storage system, general frameworks, ...

Datacenter as a Computer

Link: http://cs.wisc.edu/~shivaram/cs744-readings/dc-computer-v3.pdf

The need for a cluster of machines:

- Resource limit on a single machine → scaling out to a collection of affordable machines is generally more effective than scaling up to a giant machine
- Brings more parallelism → may also help with completing small jobs faster
- Fault-tolerance, availability, and reliability → avoid a single point of failure

Summary of trends in hardware:

- CPU speed per core is flat
- Memory bandwidth growing slower than capacity
- NVMe SSDs replacing HDDs
- Ethernet bandwidth rapidly growing

Datacenter warehouse-scale computers (WSCs) typical architecture:

- Servers → racks with a top-of-rack (ToR) switch → aggregation switch network → external Internet
- Data movement is a primary concern in the storage hierarchy
- Typically operated by a *single organization*
- Typically has *homogeneity* of resources to some extent
- Multiplexed across many applications and services for *cost efficiency*
- Implies software's support for reliability, diversity across workloads, and diversity across layers

Example workloads:

- Web indexing: latency-sensitive and bursty; use partition-aggregate scheme
- Scholar similarity: cares about throughput; not latency-sensitive; use MapReduce scheme
- Video-on-demand: transcoding is compute-intensive; cares about caching and serving
- Machine learning: long-running and extremely compute-intensive

GFS

Link: http://cs.wisc.edu/~shivaram/cs744-readings/GFS.pdf

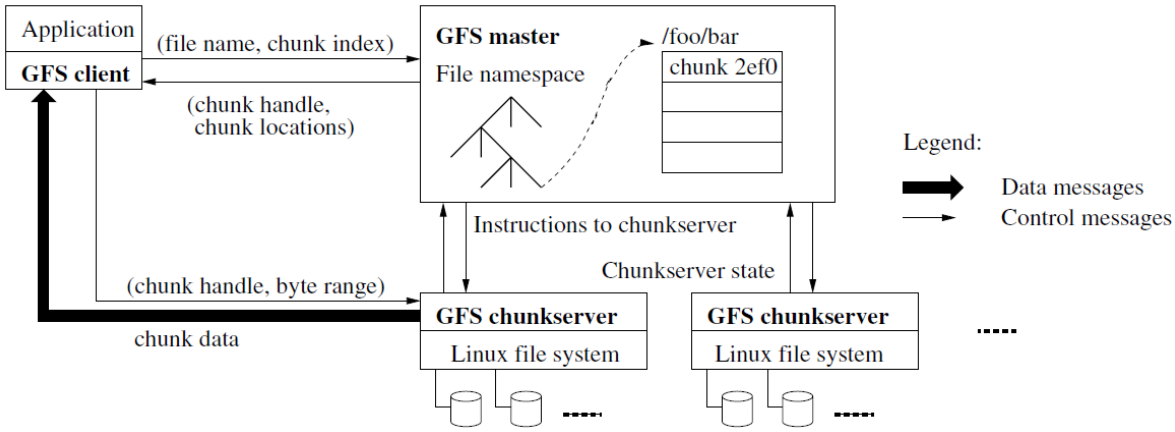
History of distributed file systems:

- Transparent POSIX API standard
- Server-client model
  - Servers run local FS on arrays of storage devices
  - Each server handles a region of the namespace tree
  - Client library sends internal requests over network
- *Caching* protocols are non-trivial
  - NFS: block-based caching in client memory; using stat timestamps for checking
  - AFS: whole-file caching on client local disk; using server callback revocations

GFS workload assumptions:

- Modest number of large files
- Reads: large streaming reads or small random reads
- Writes: large sequential appends; very few random updates
- High bandwidth more important than low latency
- Hardware component failures are a norm
- Applications can work without POSIX guarantees

GFS architecture:



- Files divided into *chunks* of 64MB
- Single *master* (possibly SMR replicated) for metadata, i.e., namespace tree and chunk location mapping
- Many *chunkservers* holding chunks as regular Linux files; uses Linux buffer cache implicitly
- Chunk size tradeoff:
  - Too small $\rightarrow$ too much metadata and control RPC load on master
  - Too large $\rightarrow$ overloading hot chunkservers and increasing fault recovery time
- Each chunk is replicated on 3 different chunkservers for fault-tolerance
  - One replica will be the *primary* replica by holding a *lease* given by master

GFS core operations & guarantees:

- `read` of a chunk
  - Procedure:

        1. client library computes which chunk to read
        2. client asks master for replica locations; master replies
        3. client goes to any replica it chooses; replica replies
        4. client caches the chunk locally until the cache lease expires or until the file is reopened
- `write` of a chunk (a *mutation*)
    - Procedure:
        1. client asks master for replica locations and who's the primary replica of chunk; master replies
        2. client pipelines written data through 3 replicas according to network topology
        3. client goes to primary replica initiating a write request
        4. primary asks secondary replicas if they have completed receiving data; secondary replicas reply
        5. chunk version number is updated on all replicas
        6. primary replies to client
    - Guarantees under concurrent requests:
        - *Consistent* -- all replicas of a chunk apply mutations in the same order assigned by primary
        - *Defined* -- if all requests span only a single chunk
        - *Undefined* -- requests spanning multiple chunks may end up with different chunks applying different mutation orders, therefore the final result is not the result of any write request
- `record append` (also a mutation, but client does not specify offset)
    - Procedure:
        1. almost the same as `write`
        2. at primary, it checks if appending the record to current chunk would cause it to exceed its maximum size; if so, it pads the chunk, tells replicas to do so, and replies to clients indicating retry on the next chunk
        3. otherwise, the primary does a normal `write` at the current end offset
        4. if a `record append` fails at any replica, the client retries the operation -- therefore, replicas may contain different numbers of the same record, thus byte-wise inconsistent
    - Guarantees under concurrent requests:
        - *At-least once* -- same record might appear more than once upon success
        - *Atomic* -- entire record will appear together
        - *Inconsistent* -- the record might appear different number of times on different replicas
- Metadata updates
    - Master maintains no directory inodes but just a full-path mapping table -- simplifies locking
    - Deletions are handled lazily and garbage collected in the background
    - Master is an SMR consensus cluster on a log of all metadata operations
    - Shadow masters are read-only, slightly lagging masters

MapReduce

Link: http://cs.wisc.edu/~shivaram/cs744-readings/mapreduce.pdf

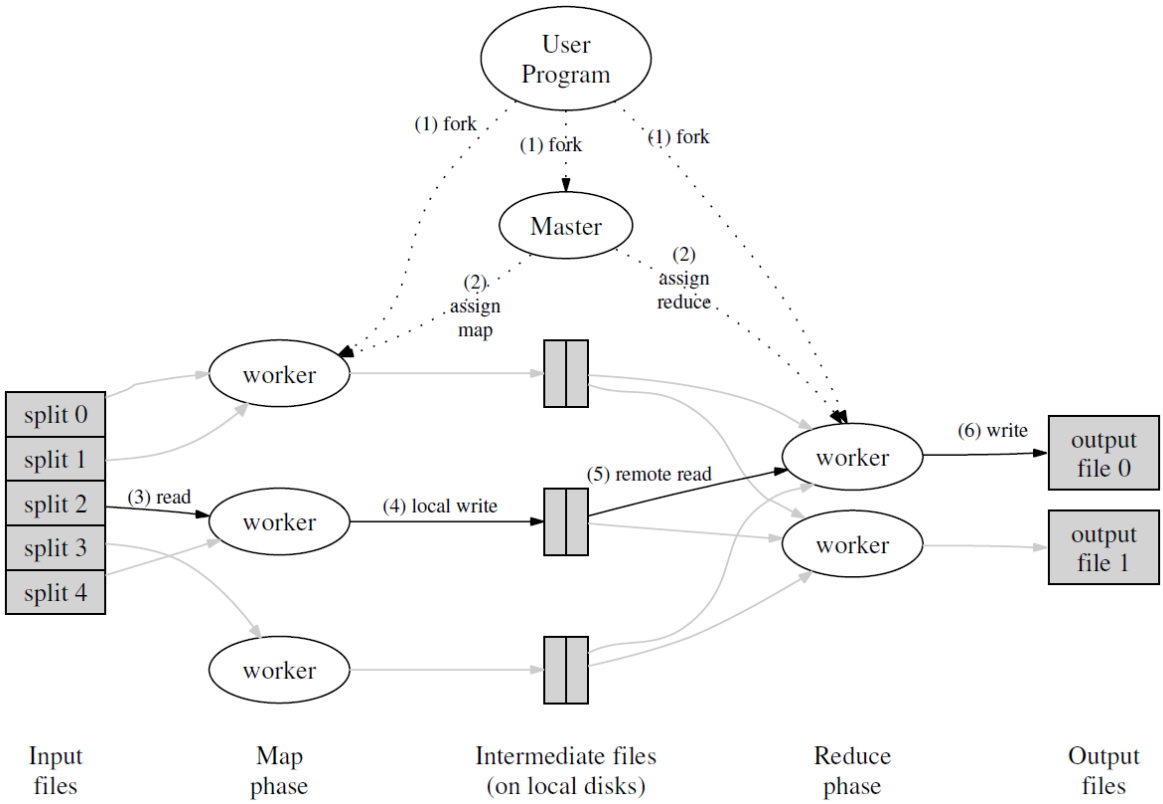
Motivation to building a general computation framework:

- Automatic parallelization of common big data tasks
- Automatic network and disk optimizations
- Handling of machine failures as a norm

Programming model:

- Data is a collection of *records*, each being a (key, value) pair
- `map` function: $(K_{in}, V_{in}) \rightarrow list[(K_{inter}, V_{inter})]$
- `reduce` function: $(K_{inter}, list[V_{inter}]) \rightarrow list[(K_{out}, V_{out})]$

Execution of the framework:



- Assumes input is a splittable collection
- Assumes local storage is fast and cheap
- Assumes replicated FS for inputs and outputs
- If a task *crashes* on a worker node:
  - Retry on another node
  - If the same tasks repeatedly fails, end the job
  - Assumes idempotent tasks with no side effects
- If a task is going slowly on a *straggler*:
  - Launch second copy of task on another node
  - Take the output of whichever finishes first
- Master failure $\longrightarrow$ simply restart the entire job

Spark

Link: http://cs.wisc.edu/~shivaram/cs744-readings/spark.pdf

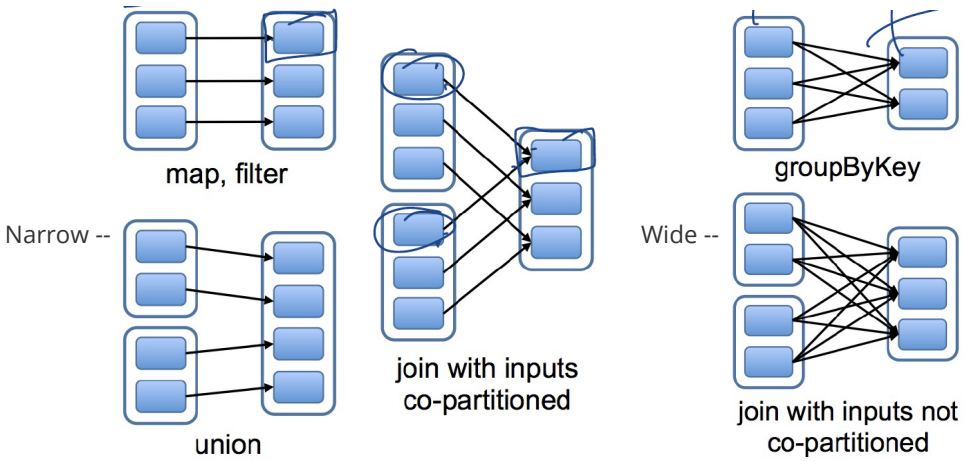
Motivation to Spark beyond MapReduce:

- *Programmability*: most real applications require multiple MapReduce steps, leading to spaghetti code
- *Data reuse* performance: MapReduce writes out massive data for every pass; expensive for *iterative* applications

Core concept: *resilient distributed datasets* (RDDs)

- *Immutable*, *partitioned* collections of (key, value) records
- May be cached (termed "persisted") in memory for fast reuse
- Supports 2 types of operations:
  - *Transformations*: RDD $\longrightarrow$ RDD(s)

    - *Narrow* transformations do not contain aggregations, e.g., `map`, `filter`, `sample`, `union`, `join` with inputs all co-partitioned with output
    - *Wide* transformations require aggregations across multiple input RDDs, e.g., `groupByKey`, `reduceByKey`, and `join` with inputs not co-partitioned with output



    - *Actions*: RDD → reduced results, e.g., `collect`, `reduce`, `fold`
- *Lineage* graph is the DAG graph of transformations used to create an RDD
    - If a worker crashed and some partition of an RDD has been lost, trace back its lineage graph and recompute that partition from what's available
    - Lineage graph can be divided into *stages*, where stage boundaries are wide dependencies; tasks within the same stage can get *pipelined*
    - Scheduler is partitioning-aware and thus avoids wide dependencies as much as possible by co-partitioning

Scheduling

Scheduling policies, cluster scheduler design and mechanisms, ...

Mesos

Link: http://cs.wisc.edu/~shivaram/cs744-readings/mesos.pdf

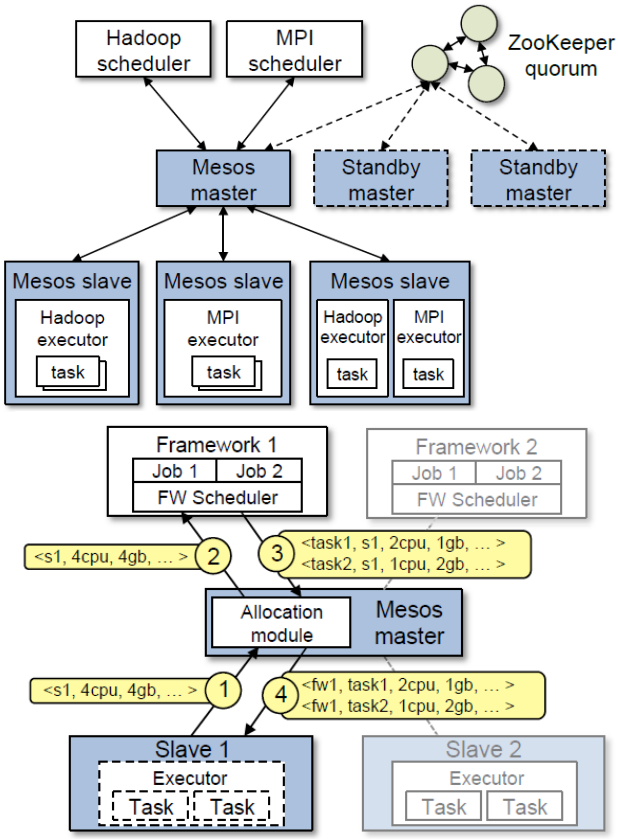
Cluster scheduling basics:

- Scheduling *policy*: algorithm or optimization problem to decide how much resource to allocate for each task
    - *Space sharing*: partition resources statically and exclusively to tasks
    - *Time sharing*: multiplex resources across multiple tasks, controlling how long does each occupy the resource
- Scheduling *mechanism*: how to implement the scheduler to enforce a chosen policy

Motivation to Mesos:

- Cluster is multiplex across multiple frameworks (MPI, Spark, MapReduce), each having its own internal scheduler across tasks
- Want to avoid rigidly partitioning the cluster across frameworks
    - may lead to under-utilization with time-varying workloads
    - may fail to elastically react to bursts
    - cannot allow data sharing and co-location of frameworks

Mesos *two-level, decentralized* scheduling architecture:



- Worker nodes periodically report amount of vacant resources through heartbeat messages
- Mesos master periodically makes *resource offers* to framework agents
    - Framework may reply with a list of tasks to allocate on those resources
    - May not use up all resources in the offer
    - Framework may also reject an entire offer due to *constraints*
        - Soft constraints: e.g., locality
        - Hard constraints: e.g., memory size
        - An optimization here is to apply *filters* before handing over offers -- resources that do not pass the filters are guaranteed to get rejected, thus don't make the offer in the first place
- Mesos master then forwards the tasks to available workers
    - Assumes most tasks are short -- when they finish, consider the next round of allocation
    - For long tasks, can do revocation beyond guaranteed allocation
- Mesos master only maintains soft state, which can be reconstructed by talking to framework schedulers and workers
- To enable placement preferences, do *lottery scheduling*: offering slots to frameworks with probabilities proportional to their preference weights on the resources

DRF

Link: http://cs.wisc.edu/~shivaram/cs744-readings/drf.pdf

Policy setting:

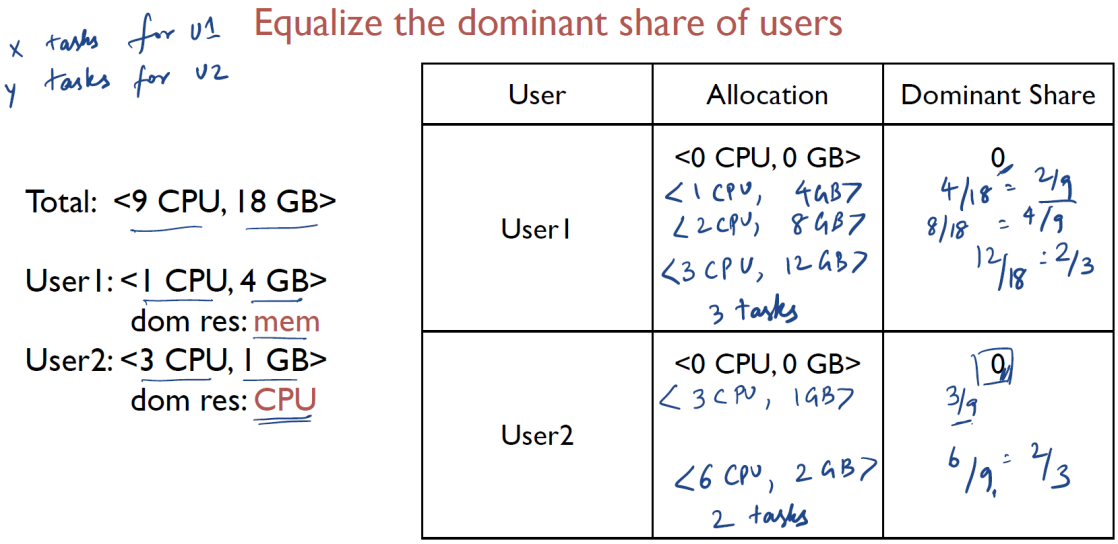
- Achieving *max-min fairness*: maximizing the minimum share across users
- *Work-conserving*: no resources are left idle
- *Slot-based* model: allocation for a task goes in units of fixed slots (called *demand vector*, e.g., $< 1 \text{ CPU}, 2\text{GB Mem} >$)
- Applications have diverse needs across multiple resources

Dominant resource fairness (DRF) properties:

- *Sharing incentive*: user is no worse off then a cluster with $\frac{1}{n}$ resources
- *Strategy proof*: user should not benefit by lying about their demands
- *Pareto efficiency*: after allocation, not possible to increase one user's allocation without decreasing someone else's
- *Envy free*: user should not prefer the allocation of another user
- *Single resource fairness*
- *Bottleneck fairness*
- *Population monotonicity*: one user leaving should not lead to decreasing other users' allocation; vice versa
- *Resource monotonicity*

DRF approach:

- The *dominant resource* of a user is the one that it has the biggest *percentage share* of
  - Value of that percentage share is called the *dominant share* of the user
  - DRF := Equalizing the dominant share of users
- Algorithm that can be used to enforce this policy: whenever there's available resource in the cluster, pick the user with the current lowest dominant share and allocate one more tasks of it



  - This algorithm provides *instantaneous fairness* and does not take into account the history of allocations

Comparison to other approaches:

- *Asset fairness*: equalize users' sum of resource shares over all resources
  - violates sharing incentive
- *Competitive equilibrium from equal incomes* (CEEI): each user receives $\frac{1}{n}$ of each resource initially; they can trade resources with other users in a perfectly competitive market
  - equivalent to maximizing the product of number of tasks over users
  - violates strategy proofness

Machine Learning

Machine learning frameworks, training systems, scheduling, serving, ...

PyTorch Distributed

Link: https://arxiv.org/pdf/2006.15704.pdf

Is an example of *data parallel* CNN training:

- Workers all hold the same complete *model* and *parameters*
- Input batch is divided into *mini-batches* and distributed across workers
- For each iteration:
  - workers compute *forward pass* on its mini-batch
  - workers compute *backwards pass* on its mini-batch to get *gradients* ∇_i
  - the global gradients (as if no parallelism) is $\nabla = \sum \nabla_i$
  - workers perform a round of *collective communication* to know the full value of global gradients
  - workers all apply the gradients to model parameters using an *optimizer*

Collective communication patterns: `broadcast`, `scatter`, `gather`, `reduce`, `allreduce`

- In CNN training, what we want to do is `allreduce`

Gradient bucketing optimization:

- Why? Because doing an `allreduce` for each individual gradient value would be prohibitively slow
  - $T_{comm} = \alpha \cdot T_{lat} + \beta \cdot T_{bw}$
- Model walks backwards the *computation graph* and assigns gradients to buckets
- When bucket becomes full, do an `allreduce` to exchange the bucket
- Can *pipeline* (overlap) backward pass computation over layers with gradient communication
- Processes can form hierarchical ProcessGroups based on network link types

PipeDream

Link: https://cs.stanford.edu/~matei/papers/2019/sosp_pipedream.pdf

Limitations of only having data parallelism:

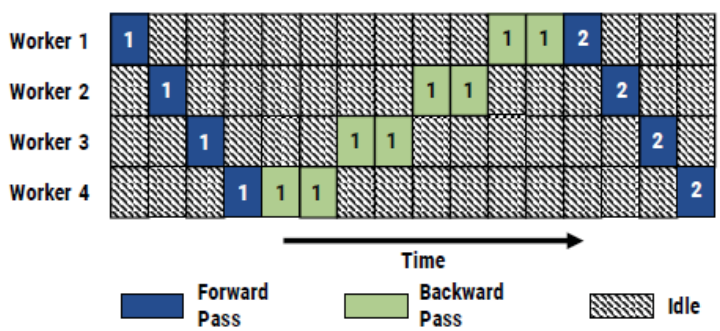
- Gradient synchronization overhead can be very high for some model types
- Overhead increases with the number of participating workers and GPUs
- Each worker has to hold the complete set of model parameters; also cannot use heterogeneous machines

Introducing *model parallelism*:

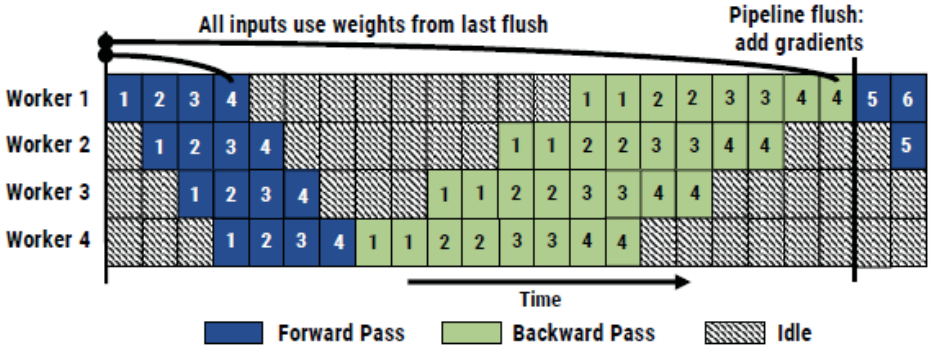
- Each worker holds a subset of model layers
- For each batch, computation flows through the workers and back

Combining the two parallelism paradigm gives us *pipeline parallelism*:

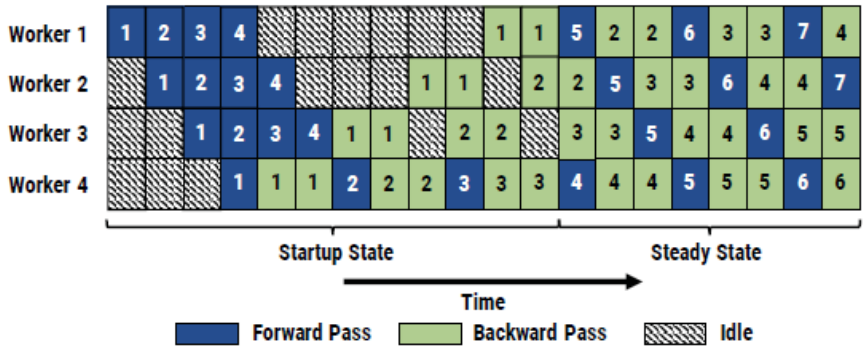
Vanilla model parallelism --



GPipe pipeline parallelism --



PipeDream pipeline parallelism --



- Instead of feeding 1 input batch each round, feed in k batches and pipeline their computation; Steady state utilization should be high

Challenges:

- Stage balancing: want stages to have the same amount of work
  - Solution: stage can be replicated, e.g., having two workers do the first stage, etc.
  - Use a profiler to estimate the computation time of forward & backward of each layer, their size of parameters, etc., do the best split according to profile
- Workload scheduling:
  - Solution: always do 1 forward 1 backward in steady stage on a worker (1F1B)
- Effective learning: in PipeDream's version of pipeline parallelism, the forward pass of the next batch is not using the most up-to-date model parameters
  - e.g., the forward of 5 uses a model where only the gradient of batch 1 is applied, not 2~4
  - Introduces a stochastic *convergence* problem (i.e., *model staleness*)
  - *Weight stashing*: workers also have to keep several recent versions of the model so that on-flight forward passes of a batch use the same model version across workers
    - Loses the benefit of memory overhead saving due to this mitigation

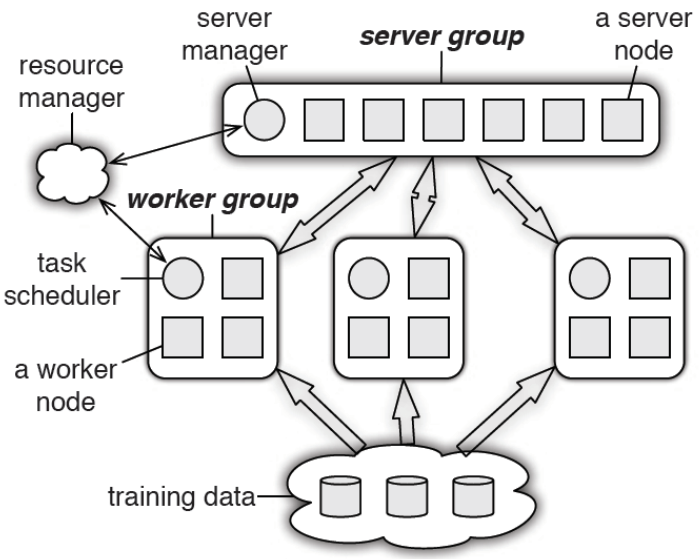
Parameter Server

Link: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-li_mu.pdf

Motivation to parameter server design:

- Model is getting excessively large (up to 10^{12} parameters), hard to hold on computation nodes
- Efficient and flexible gradient synchronization
- Fault-tolerance and durability of intermediate model state
- Most model parameter updates are *sparse* -- most entries are zeros, so updates can be compressed
- Scalability of model storage and optimizer updates

Parameter server architecture:



- Model is partitioned and stored (with replication) on parameter server nodes
- Typical round of iteration:
    1. workers do backward pass, push (sparse) gradients to servers
    2. servers apply gradients with an optimizer to update the model parameters
    3. workers pull parameters from servers for the next forward pass
- API supports range push and pulls, as well as user-defined functions
- As discussed before, can overlap iterations at the cost of having model staleness

"Consistency" (staleness) models:

- Sequential: always synchronize at iteration boundaries -- always up-to-date
- Eventual: no guarantee on parameter freshness
- *Bounded delay*: parameter is bounded to be at most k iterations stale

Implementation details:

- Parameter versions implemented using *vector clocks* associated with each parameter
    - on every push, increment the timestamp at the workers index in the vector clock
    - on every pull, returns vector lock alongside
- Model partitioned & replicated across servers using *consistent hashing* (DHT)
    - immediately-next node on ring is the primary replica
    - the following several nodes are secondary replicas for this partition
- Straggler workers can be either replaced by adding a new worker or simply terminated

Gavel

Link: https://www.usenix.org/conference/osdi20/presentation/narayanan-deepak

Motivation to Gavel:

- Hardware *heterogeneity*:
    - Different types of accelerators, memory sizes, etc.
    - Different models have different preferences to accelerator type
- Supporting a wide range of objectives:
    - Minimize makespan
    - Minimize average job completion time (JCT)
    - Fairness

- Placement/co-location sensitivity

Heterogeneous scheduling is an optimization problem with the following constraints:

$$0 \leq X_{mj} \leq 1 \qquad \forall (m,j) \quad (1)$$
$$\sum_j X_{mj} \leq 1 \qquad \forall m \quad (2)$$
$$\sum_m X_{mj} \cdot \text{scale\_factor}_m \leq \text{num\_workers}_j \qquad \forall j \quad (3)$$

$$X^{\text{example}} = \begin{array}{c} \begin{matrix} V100 & P100 & K80 \end{matrix} \\ \begin{pmatrix} 0.6 & 0.4 & 0.0 \\ 0.2 & 0.6 & 0.2 \\ 0.2 & 0.0 & 0.8 \end{pmatrix} \end{array} \begin{array}{l} \text{job 0} \\ \text{job 1} \\ \text{job 2} \end{array}$$

- m: model, i.e., task
- j: accelerator type
- X: the allocation matrix

The *objective* of the optimization problem depends on the chosen scheduling policy:

- Weighted max-min fairness: $\text{Maximize}_X \min_m \frac{1}{w_m} \frac{\text{throughput}(m,X)}{\text{throughput}(m,X_m^{\text{equal}})} \cdot \text{scale\_factor}_m$
  - where $\text{throughput}(m,X) = \sum_j T_{mj} \cdot X_{mj}$, i.e., weighted sum over allocation
  - throughput matrix T comes from the *estimator* module or provided by user
- Minimize makespan: $\text{Minimize}_X \max_m \frac{\text{num\_steps}_m}{\text{throughput}(m,X)}$
- Minimize finish-time fairness: $\text{Minimize}_X \max_m \rho_T(m,X)$
  - where $\rho_T(m,X) = \frac{t_m + \frac{\text{num\_steps}_m}{\text{throughput}(m,X)}}{t_m^{\text{isolated}} + \frac{\text{num\_steps}_m}{\text{throughput}(m,X^{\text{isolated}})}}$
- FIFO: $\text{Maximize}_X \sum_m \frac{\text{throughput}(m,X)}{\text{throughput}(m,X^{\text{fastest}})}(M-m)$
  - where M is the total number of tasks
- Shortest job first: $\text{Minimize}_X \min_m \frac{\text{num\_steps}_m}{\text{throughput}(m,X)}$
- Minimize total cost: $\text{Maximize}_X \frac{\sum_m \text{throughput}(m,X)}{\sum_m (\sum_j \text{cost}_j \cdot X_{mj})}$
- Can have *hierarchical policies*, where each node may deploy a different policy across its children nodes
  - Solve an optimization problem across entire organization
  - where weights are constrained by policy within entity, and multiply-chained

Scheduling mechanism -- *round-based* scheduler:

- Every round is ~6 mins
- At the beginning of every round, consider a list of schedulable jobs and solve an optimization problem to get X

- Decide which jobs are chosen to run on which accelerators in this round, based on their number of rounds spent on each accelerator type
  - compute all jobs' priority on each accelerator type by $\frac{X_{mj}}{\text{rounds\_recv}_{mj}}$
  - for each accelerator, schedule the job with the highest priority on it
- Is a greedy mechanism -- converges over rounds

Nexus

Link: https://dl.acm.org/doi/pdf/10.1145/3341301.3359658

Inference workloads and video analysis serving:

- Jobs have *latency service-level objectives* (SLOs) that must be met to be considered successful
- Lots of data coming in interactively and continuously
- Each stream is processed by a DNN forward-pass "query"

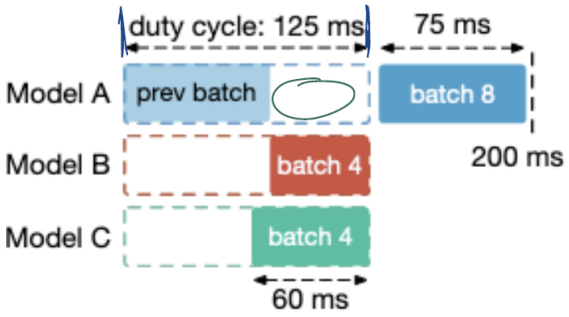
Scheduling goal is to have high GPU *utilization*:

- Deciding which GPUs hold which ML models
  - moving models to GPUs is expensive operation
  - GPU memory is limited, cannot hold all models at once
- Job *batching* improves utilization but has to control its latency: $\text{batch\_lat}(b) = \alpha b + \beta$
  - batch too small $\rightarrow$ low latency for that single job, but low utilization
  - batch too big $\rightarrow$ high latency, may miss latency SLOs, but high utilization

Batch-aware scheduling procedure:

- Do profiling etc. to get a batch-to-performance mapping table as the following example
- Objective is reaching the target throughput for all models, while meeting the latency SLO
- Idea: divide time into *duty cycles*, length = $\frac{1}{2}$ of latency SLO
  - Why? Because need to ensure that an arriving task that just misses the start of a cycle can catch the next cycle and still meet its SLO
  - From table, for each model, choose the batch size configuration that at least reaches its throughput target
  - Then, try to pack them into the duty cycle

Model A			Model B			Model C		
Batch	Lat	Req/s	Batch	Lat	Req/s	Batch	Lat	Req/s
4	50	80	4	50	80	4	60	66.7
8	75	107	8	90	89	8	95	84
16	100	160	16	125	128	16	125	128



- How to do the packing exactly?
  1. given each model's input *request rate*, find the batch size configuration with the best throughput, and start with k full GPUs s.t. $k \cdot T_{best} \leq$ request rate (so that we won't stall on client input)

2. take the minimum duty cycle length across all models
3. try to pack two batches into one duty cycle -- if does not fit, try reducing the batch size of one model until fit (so that we can pack, though that model's throughput may not be the optimal any more)

Complex queries involve inference along a sequence of models:

- Need to break down end-to-end SLO into individual model SLOs
- Need to do query analysis to determine request rate splits if there are branches in the graph

Adaptive batching:

- Clipper adapts batch size dynamically according to the request at head of queue
  - may lead to a high fraction of SLO misses and low utilization
  - *early-dropping* requests that cannot miss SLO is a good idea
- Nexus does batch-aware dispatch: after batch size has been decided, do *sliding-window* over queue, until requests in the entire window can meet SLO
  - requests ahead of them are just dropped, since they cannot meet SLO anyway with this batch size

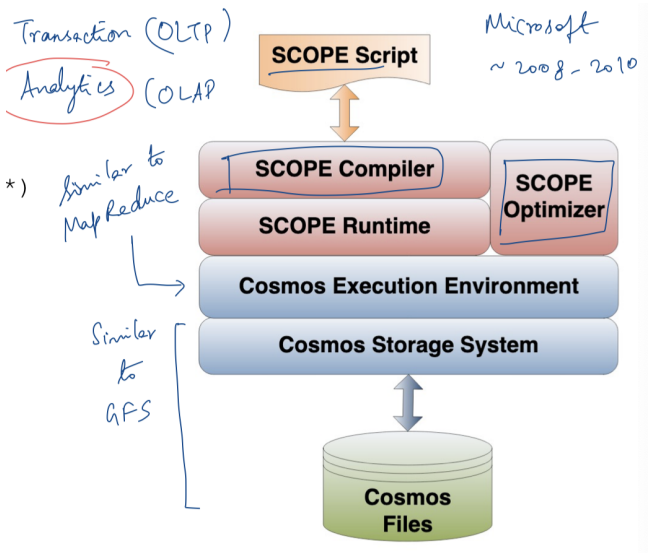
SQL Frameworks

Database analytical frameworks for SQL, ...

SCOPE

Link: http://www.vldb.org/pvldb/vol1/1454166.pdf

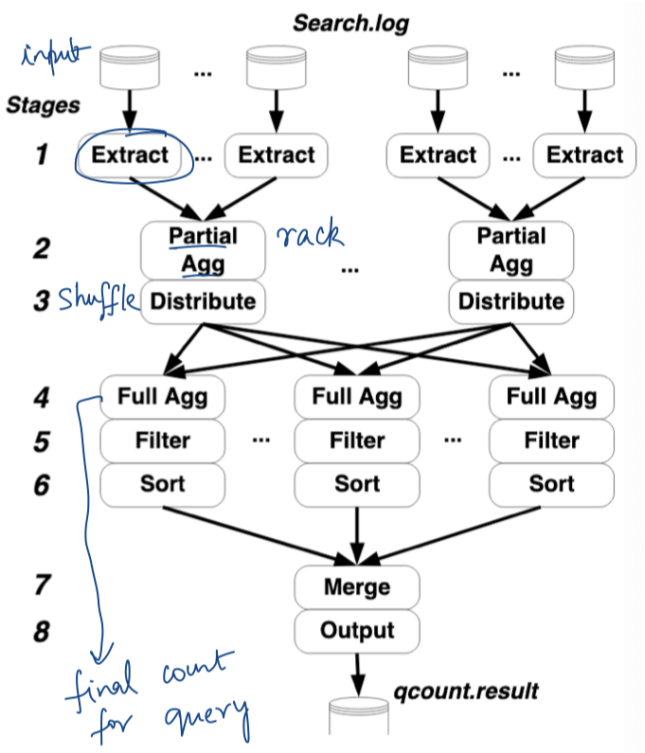
SCOPE architecture for analytical workloads in SQL (*Structured Query Language*):



- Query language extension to SQL:
  - Input reading (*Extraction*): input can be from many sources; allows custom extractor function specifying the logic for getting one row
  - All common SQL operators are supported
  - *Language integrated queries* (LINQ): allows calling C# functions inside the queries
- Execution pipeline is MapReduce-like:
  - *Process - Reduce*
  - *Combine* is new: takes multiple tables that are co-partitioned, outputs one table

System components:

- *Compiler*: parses query and returns an internal parse tree
- *Optimizer*: rewrite the query parse tree to get a *query plan* with the lowest estimated cost
    - Examples:
        - Removing unnecessary columns
        - Push down selection predicates
        - Pre-aggregating
    - Query plan is aware of being in a distributed system:



- *Runtime*: applies run-time optimizations, e.g., locality-aware task placement

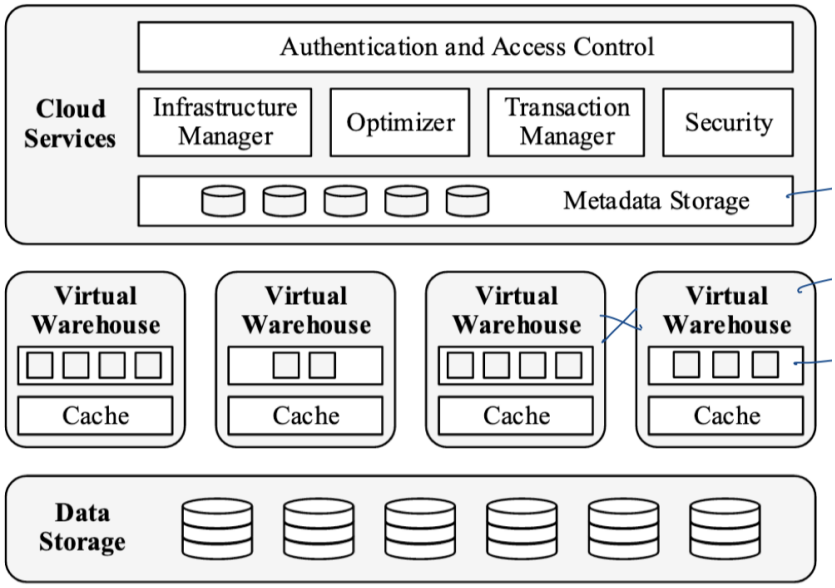
Snowflake

Link: https://event.cwi.nl/lsde/papers/p215-dageville-snowflake.pdf

Snowflake is a cloud-native, *storage-disaggregated*, *elastic* data warehouse, which enables *data analytics as a service*; goals:

- Software-as-a-Service
    - Considers mostly read-only analytical workloads
- Storage disaggregation
    - Shared-nothing: cannot independently scale compute and storage, but has good locality
    - Shared-data via disaggregated storage: can scale independently and can leave data replication entirely in the storage service, but loses locality
- Elastic scaling according to workload
- Highly available
- Semi-structured data (partially data lake)

Snowflake architecture:



- *Hybrid-columnar* format for data storage
- *Virtual warehouses* (VWs) are entirely stateless and isolated among users
  - Deploys local *caching*
  - To handle stragglers, does *work stealing*
  - A query may span multiple VWs
- *Cloud services* are a collection of services for metadata & control
  - Metadata storage on a fast cloud key-value store
  - Transaction manager: MVCC concurrency control
  - Query optimizer: min-max pruning, etc.
- For fault-tolerance, services and storage span multiple datacenters worldwide
  - Storage is the ground-truth and is geo-replicated
  - Each VM resides in one datacenter and is stateless
  - Each cloud service is stateless

Stream Processing

Streaming, events, dataflow, ...

Dataflow Model

Link: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43864.pdf

Streaming data means *unbounded* data that may arrive *out-of-order*, e.g.:

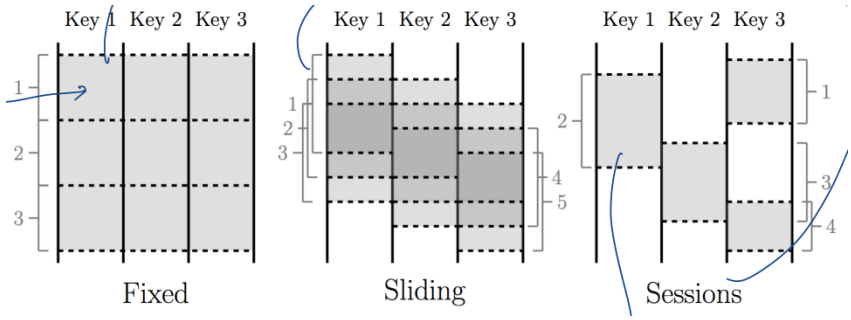
- Physical sensor readings
- Logs generated by services
- Shared service with user sessions (e.g., multi-player gaming)

Streaming vs. Batching:

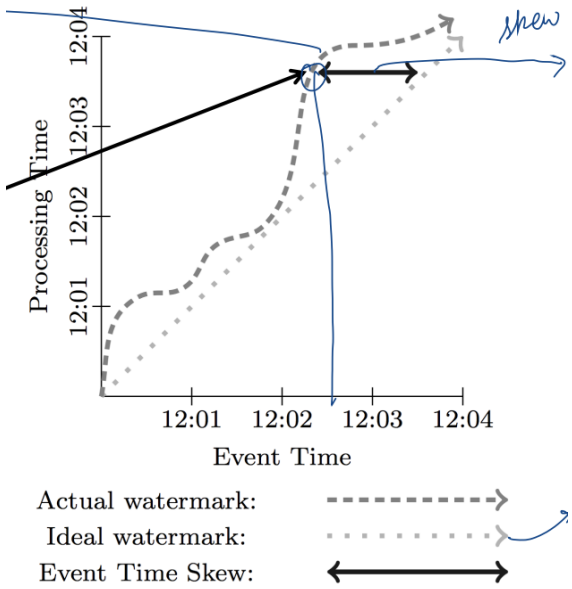
- Previous *batch* processing can be viewed as a special case of stream processing, where we wait for a fixed amount of time or input data, then run a batch job (e.g., MapReduce or Spark) to calculate the output for that batch.
- In streaming, we must differentiate the following types of timestamps:
  - *Event time*: time at which an event happened, tagged in the event
  - (*Arrival time*: time at which an event arrives at the system)
  - *Processing time*: time at which the event is used in computation
  - Processing time $\geq$ Arrival time $\geq$ Event time

Dataflow model concepts:

- *Windowing* (on event time):

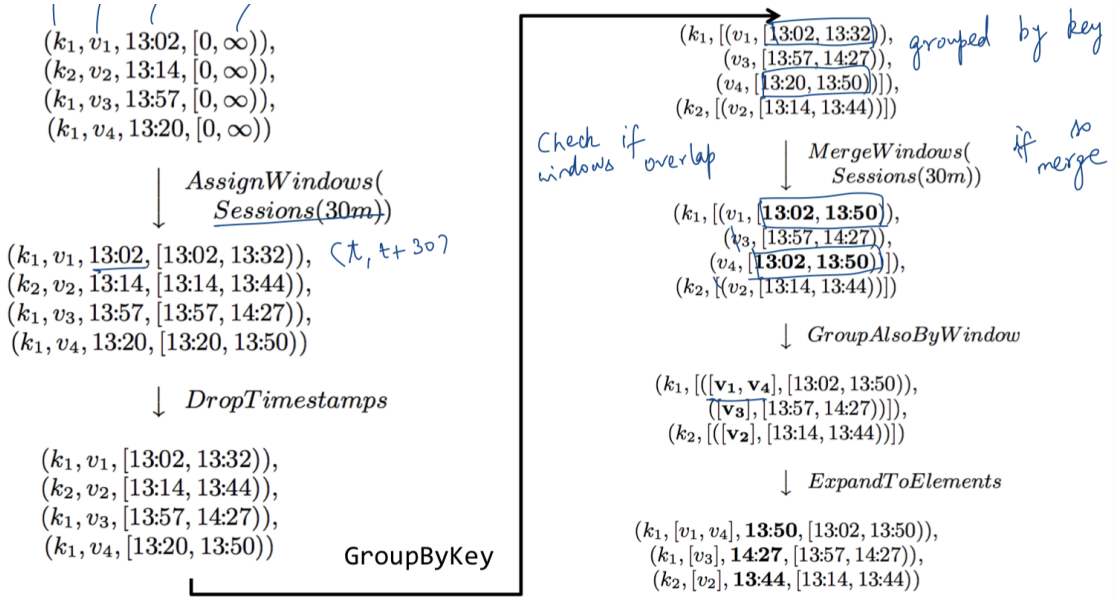


  - *Fixed* (or *Tumbling*)
  - *Sliding* (or *Moving*)
  - *Sessions* (activity triggered/grouped)
- *Watermark* that represent the *skew* (or *lag*) of system processing:



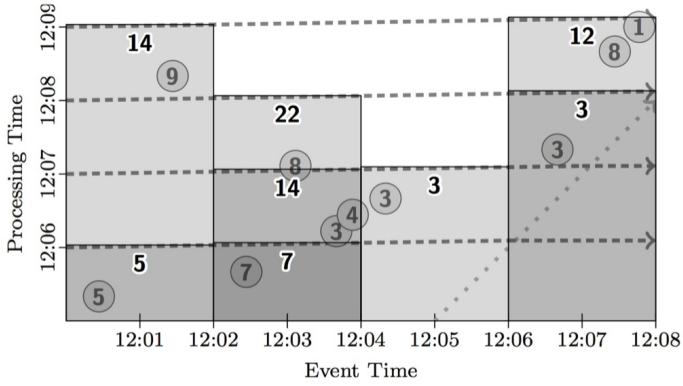
  Note that the watermark is assumed to be an input given to the system as a heuristic

- Data processing API:
  - `ParDO`: map
  - `GroupByKey`: group events by key
  - `AssignWindow`: put event into window(s)
  - `MergeWindow`: useful for sessions

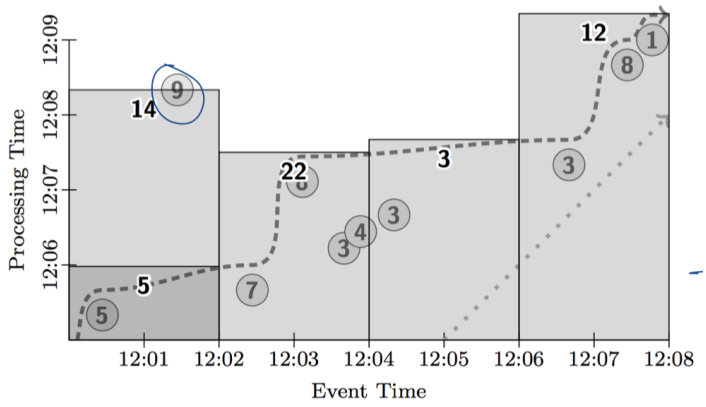


- *Triggering*: when in processing time are groups emitted
  - Trigger types:
    - At fixed time period
    - At fixed count of arriving events
    - At given watermark
  - Accumulation strategies:
    - *Discarding*: discard old sum
    - *Accumulating*: accumulate on old sum
    - *Accumulating & Retracting*: accumulate on old sum, while also returning negative old sum
  - Example of fixed windows, micro-batch watermark:

```
PCollection<KV<String, Integer>> output = input
    .apply(Window.into(FixedWindows.of(2, MINUTES))
                .trigger(Repeat(AtWatermark()))
                .accumulating())
```



  - Example of fixed windows, arbitrary watermark:

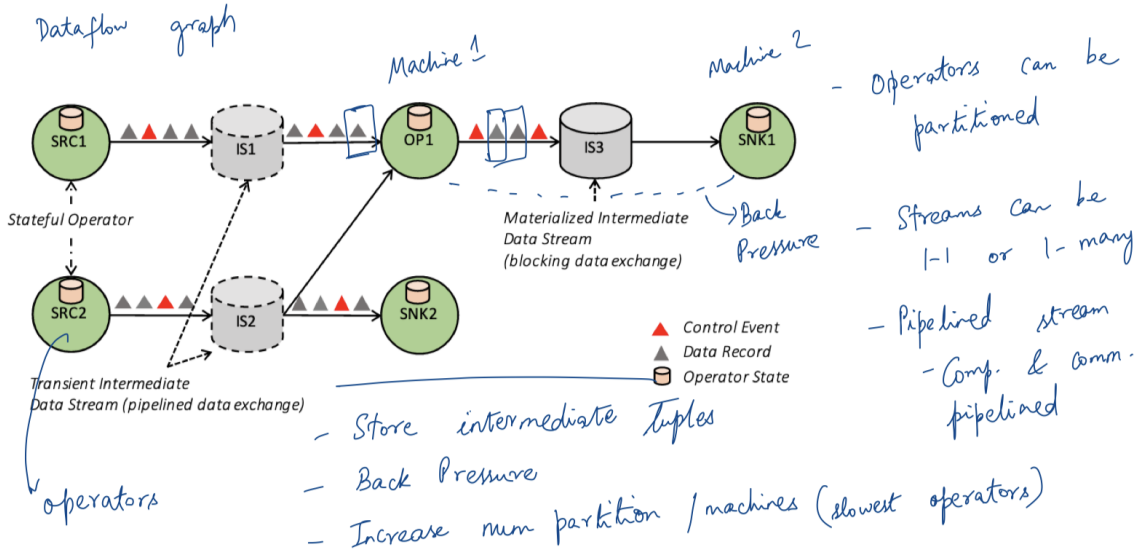


Apache Flink

Link: https://asterios.katsifodimos.com/assets/publications/flink-deb.pdf

Flink's computation model:

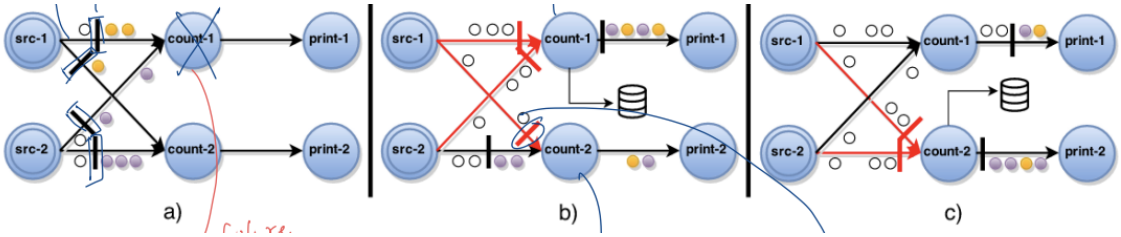
- Long-lived *operators* that implement map/reduce/windowing
  - Operators can be partitioned (scaled to multiple machines)
  - Operators can be *stateful*: they internally maintain mutable state
    - Windowing
    - Aggregation, running sums, etc.
- *Intermediate data streams*:
  - Transient: pipelined data push
  - Materialized: blocking data push, materialized on disk



- A special batch processing mode is integrated
  - Allows new blocking operators, e.g., `sort`, that can only be used in this mode

Fault-tolerance techniques to recover from failed stateful operators:

- *Asynchronous barrier snapshotting* to periodically snapshot operators state to persistent storage



  - By injecting a "start snapshot" control message barrier to the dataflow
  - An operator makes snapshot at the time when it receives snapshot control messages from all of its input edges -- before that it must block on edges that it has received a control message from
  - Assumes *in-order* message delivery
- Assumes that the input data source is *replayable*, e.g., Apache Kafka
- After failure, reset all operators to snapshot, redo all processing from there beyond

Spark Streaming

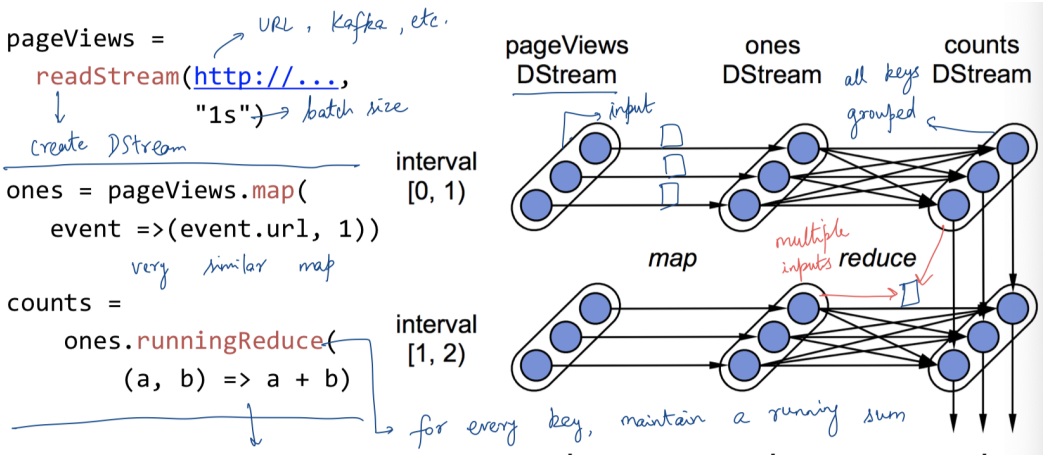
Link: https://people.csail.mit.edu/matei/papers/2013/sosp_spark_streaming.pdf

Operator fault-tolerance in previous streaming frameworks:

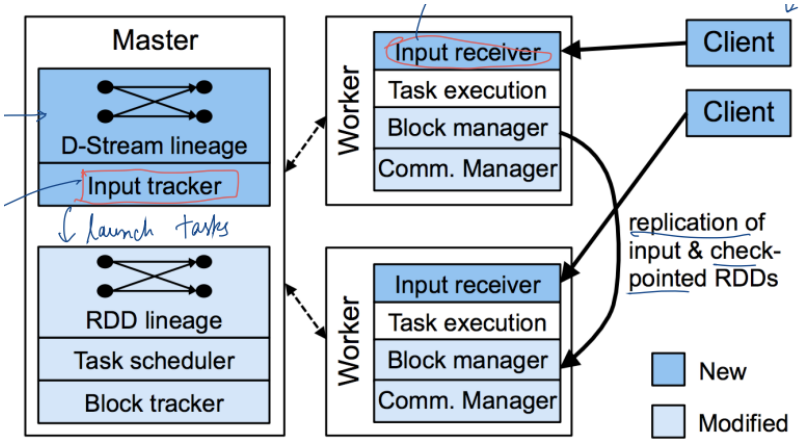
- Replicated operators: higher resource consumption; higher synchronization overhead
- Snapshotting + replaying: high fail-over cost

Discretized streams (D-Streams) to mitigate these problems:

- Divide time into fixed intervals
- Run Spark-like stateless batch job on each discredited interval
  - Output and state are (persistently) saved
  - State can be tracked as streams of (key, event) ⟶ (key, state)
- APIs:
  - All stateless Spark transformations
  - Stateful windowing operations: `slidingWindow`, `reduceByWindow`, etc.
    - Associative only vs. Associate & invertible sliding window
- Example of running sum:



Spark streaming system architecture:



- Optimizations:
  - Timestep pipelining: no barrier across time steps unless needed (faithful to the lineage graph)
  - Checkpointing: checkpoints are done by async. I/O as RDDs are immutable; truncate lineage graph after checkpoint
  - Stragglers are mitigated by speculative execution
- Fault-tolerance: *parallel recovery*
  - When a worker fails, we need to recompute state RDDs stored on worker, then re-execute tasks running on the worker
  - Strategy is to run all independent recovery tasks in parallel on all other healthy worker machines in the cluster

Graph Processing

Iterative graph algorithms, embedding learning, graph neural networks, ...

PowerGraph

Link: https://www.usenix.org/system/files/conference/osdi12/osdi12-final-167.pdf

Background on classic graph analytics:

- Examples:
  - PageRank
  - Shortest path
  - Connected components
- *Pregel* programming model: *Vertex programs*
  - "Think like a vertex"
  - Receive inputs from neighbors, combine $\rightarrow$ Do computation on vertex $\rightarrow$ Send message to neighbors
- *Natural graphs* impose significant challenge to existing frameworks:
  - Exponential distribution of neighborhood sizes
  - Lack of symmetry in graph structure, hard to perfectly partition

PowerGraph's *Gather-Apply-Scatter* programming model:

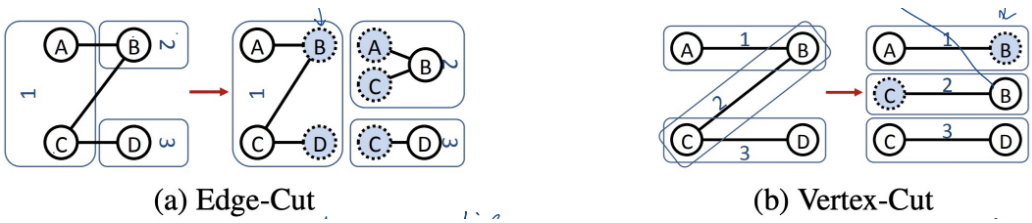
- Gather: function called on receiving in-edge information from a neighbor
- Apply: apply accumulated result of gather to vertex state
- Scatter: send update to an adjacent neighbor
  - May possibly *activate* the neighbor if the difference (delta) is sufficiently large

Execution model:

- Active vertices:
  - At the beginning, all vertices are activated
  - In scatter, activate destination neighbor for the next iteration only if the update is sufficiently significant
- Accumulators and vertex states are persistent across iterations
- *Delta caching*:
  - Cache accumulator value computed in previous iteration for reuse
  - Scatter optionally returns a delta; accumulate only the deltas to avoid running Gather for that vertex
- Synchronous vs. Asynchronous execution:
  - Sync: barrier after each *minor-step* (e.g., all the Gathers of an iteration)
  - Async: no barriers; execute active vertices as cores become available
    - Can optionally choose serializable mode: connected vertices are not processed concurrently

Graph *partitioning* for distributed execution:

- Edge-cut vs. Vertex-cut:



  - *Edge-cut*: previous systems assign vertices to machines; neighbors at the other end of a cut edge become a *ghost vertex* $\Rightarrow$ imbalance for natural graphs
  - *Vertex-cut*: assign edges to machines; when a vertex is on many machines, one is the *primary* of the vertex and data is synchronized across replicas
- Edge distribution policies:
  - Random placement
  - *Coordinated greedy* placement: if either vertex of an edge is already placed, favor those machines
  - *Oblivious greedy* placement: avoid coordination during parallel partitioning by only tracking vertices present locally

Marius

Link: https://www.usenix.org/system/files/osdi21-mohoney.pdf

Graph embeddings learning:

- Given a graph structure, learn a vector representation (i.e., embedding) for each vertex such that it captures the graph structure
  - Then we can do efficient (approximate) K-nearest-neighbor algorithms, etc., using those embeddings
- *Score function* on an edge: $f(emb_{src}, emb_{dst}) \rightarrow score$
  - Score can be the opposite of some distance function
  - We want to maximize scores for edges in graph
  - At the same time, minimize scores for non-existing (negative) edges
  - Loss $L = \sum_{e \in G} \sum_{e' \notin G} \max(f(e) - f(e') + \lambda, 0)$

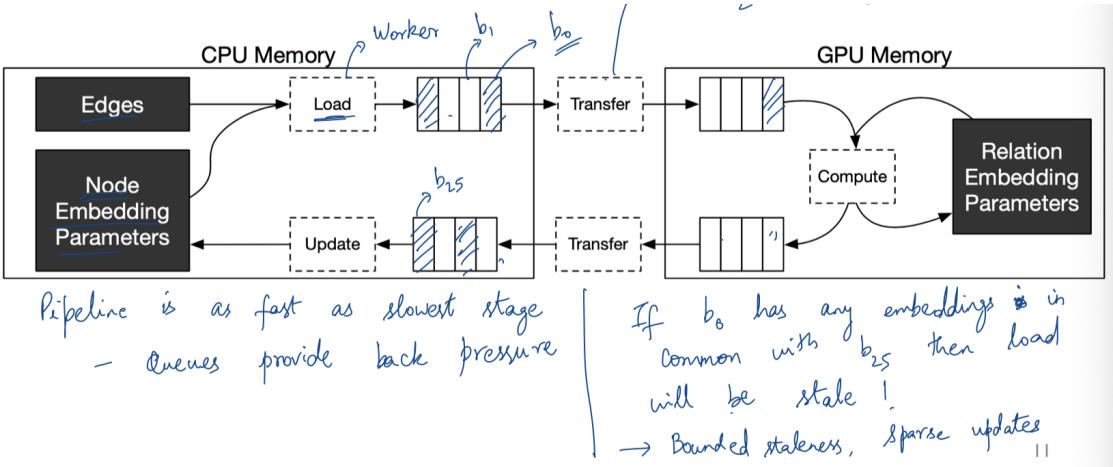
- Training algorithm can be standard SGD/AdaGrad optimizer

```
for i in range(num_batches):
    B = getBatchEdges(i)
    E = getEmbeddingParams(B)
    G = computeGrad(E, B)
    updateEmbeddingParams(G)
```

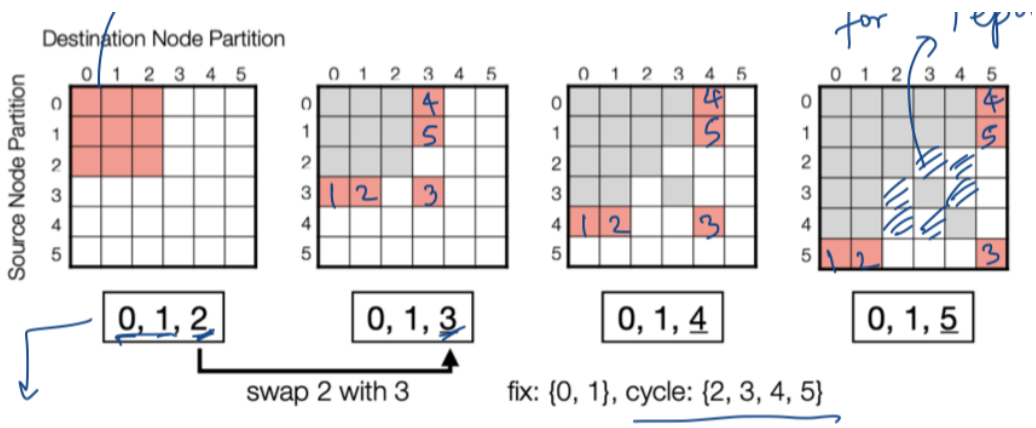
- Note that this "learning" here has no concept of "model" or "training vs. testing" like in neural networks: our goal here is solely to generate good vertex embeddings for a known graph
- Large graphs lead to excessively large embedding sizes, cannot fit in memory
    - Data movement between CPU-GPU memory or between memory-disk is a major challenge

Marius design for I/O-efficient graph embeddings learning:

- Pipelined training:



    - If the graph has relation embeddings (i.e., edge type embeddings) as well (which tend to be small), store those in GPU memory
    - Pipelining introduces *staleness*: if a pre-loaded batch has vertices in common with a just-computed batch that has not been updated on CPU memory, those embedding values will be stale
- Out-of-memory training:
    - Maintain a cache of graph partitions (partitioned by edges) in CPU memory
    - To traverse partitions in a cache-friendly way, uses *BETA-ordering*:



- Initialize cache with c partitions
- In each cycle, keep the first $c-1$ partitions and swap in one next partition
- Once all partitions touched once, re-initiate cache and repeat

DistDGL

Link: https://arxiv.org/pdf/2112.15345.pdf

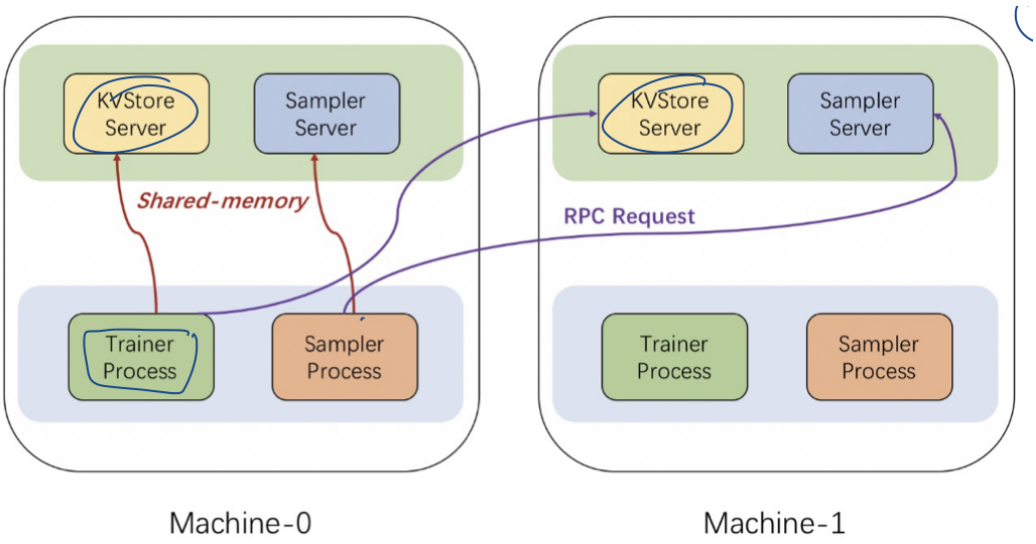
Classic graph embedding learning vs. Graph neural networks (GNNs):

- Embedding learning in Marius is *decoder-only*
- GNNs, however, use a neural network to capture neighborhood structure

$$h_i^k = AGG_W(h_i^{k-1}, \{h_u^{k-1} : u \in N_i\})$$

  - N_i is the one-hop neighborhood of vertex i
  - AGG is a model-parameterized aggregation function
  - Go through l such layers of a GNN effectively aggregates information of l-hop neighbors for each vertex

Distributed Deep Graph Library (DGL) system overview:



- KVStore stores current embedding state
  - Apply METIS to partition graph across machines
  - Re-apply METIS to partition within a machine
  - Adopts heterogeneous graph partitioning to support knowledge graphs with multiple edge types (have them co-partitioned)
- Sampler samples the graph structure into mini-batches for training
  - Start with a set of b target vertices
  - Sample n neighbors of current vertices, resolve common neighbors
  - Repeat expansion for some iterations
- Trainer runs NN training
  - Async mini-batch sampling with sync training: sample neighborhood and embeddings for future mini-batches asynchronously only if current batch is not updating them

New Models

Serverless computing, content distribution, TPUs, ...

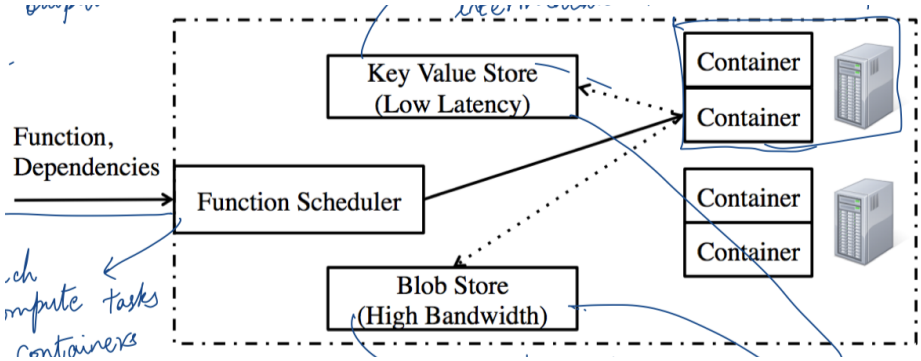
Serverless Computing

Link: https://shivaram.org/publications/pywren-socc17.pdf

Usability is the main motivation behind serverless computing:

- Users write simple, *stateless* computation functions
- The framework completely abstracts away hardware management, scaling, fault-tolerance, execution models, etc.
- "Cloud button"
- Network I/O cost from/to a storage service is now is competitive speed with local SSDs

PyWren state data processing architecture:



- Functions (operators) cannot save state; input/output must be through external storage
  - Key-value store for low latency, small volume
  - Blob storage for high bandwidth
- Cluster-side scheduler schedules functions as virtual machines or containers to run in a cluster
  - Does not have to cold-start containers each time; may have a pool of standby containers
- Function dependencies are tracked on application side library

When to use serverless:

- Yes cases:
  - Tasks are independent
  - Latency is not a concern
  - Cluster utilization is low
- No cases:
  - Long running tasks with checkpoint overheads
  - Iterative algorithms with intermediate state
  - Convoluted function dependencies

Owl

Link: https://www.usenix.org/system/files/osdi22-flinn.pdf

Content distribution workload:

- Distribution of read-only content to a huge amount of clients
  - Website resources
  - Video-on-demand
  - Docker container binaries
  - AI model parameters
  - Search indexes

- This is different from previous workloads discussed:
    - Read-heavy (-only), very few internal updates (typically with multi-versioning applied)
    - Can be extremely skew towards hot content
    - Number of clients can scale to millions or more
    - Different content may want different policies

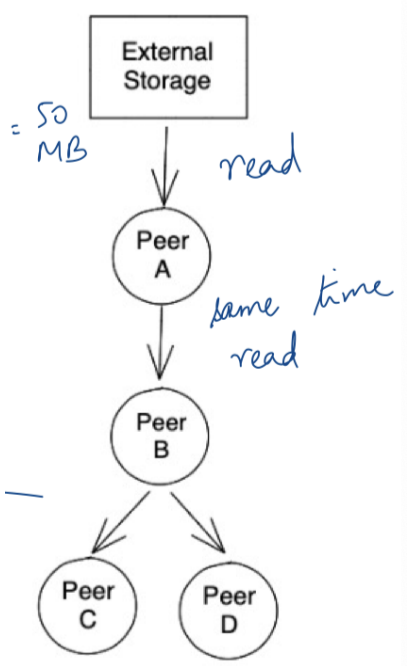
Previous solutions to content distribution:

- Hierarchical caching: deployed in most CDN solutions
    - Centralized solution, need centralized control over all cache servers
    - Requires a ton of resources to scale to millions of clients
- Peer-to-peer (P2P) exchange: e.g., BitTorrent
    - Scalable and decentralized
    - Inevitable staleness in data
    - Lack of global picture, policy enforcement, and manageability

Owl system design & architecture:

- Data source is assume to be a highly-available, scalable storage service
- *Peers* run in each client-side library inside applications, forming the data plane
    - Simple API; asks (a random) tracker where to fetch data and whether to cache it in memory/disk
    - Assumes clients are still organizational users whose resources can be used for caching
    - *SuperPeers* are standalone peers not linked into client applications, providing more reliable resources for caching
- *Trackers* store metadata mappings (peers ↔ cached chunks) and controls which peers cache what data + which peers fetch data from where
    - Soft state, similar to GFS
    - A tracker will internally be a replicated service for fault-tolerance

Owl uses the idea of *ephemeral distribution trees* to maintain a view of what is going on:



- Edges represent on-going data transfer, hence ephemeral
- Policies affect decisions made by tracker:

    - *Selection policy*: upon receiving a new fetch request from a peer, decide which peer (or external storage) should it fetch data from
    - *Caching policy*: which blocks should the peer keep in its memory/disk
    - Policies can be different across different data and applications
- To avoid trackers becoming bottlenecks, peers are sharded and may belong to different tracker
    - Trackers periodically exchange metadata
    - A tracker can decide to *delegate* a peer's request to another tracker to let it make decision

Disadvantages of Owl's design:

- Client network bandwidth is used by Owl's content distribution
- For very small files, latency added by RPCs to tracker can be significant
- Client needs to sacrifice memory/disk resources for caching
- Peer churn could be high if applications are short lived

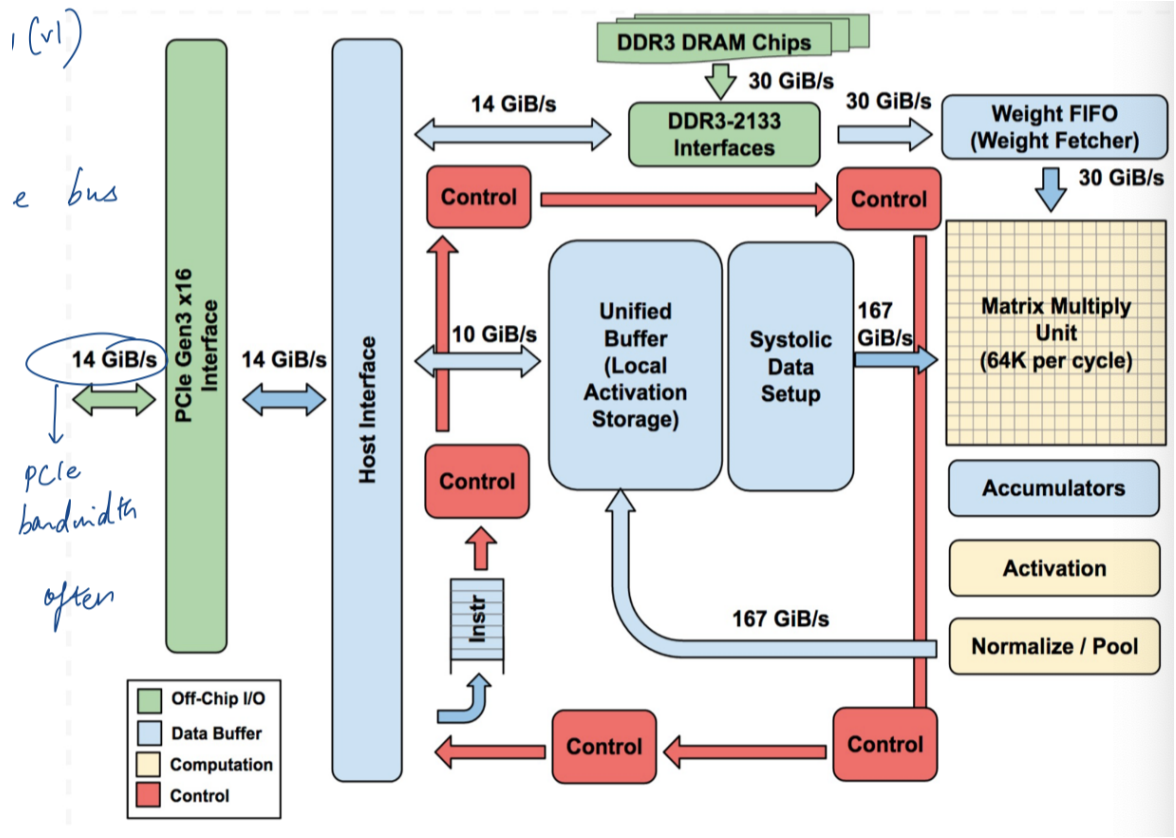
TPU

Link: http://stanford.edu/class/cs114/readings/tpu.pdf

Motivations behind building specialized hardware accelerator for inference:

- New products (voice assistants) drastically increase inference workload volume
- Energy consumption (power) per operation becomes a critical metric
- Total cost of ownership of high-end generic CPUs/GPUs are high
- ML inference workloads have potentials in building simplified hardware
    - *Quantization* to lower precision saves time and energy
    - 8-bit integer multiplications, not floating-point
    - Need for predictable latency but not throughput

Tensor Processing Unit (TPU) architecture:

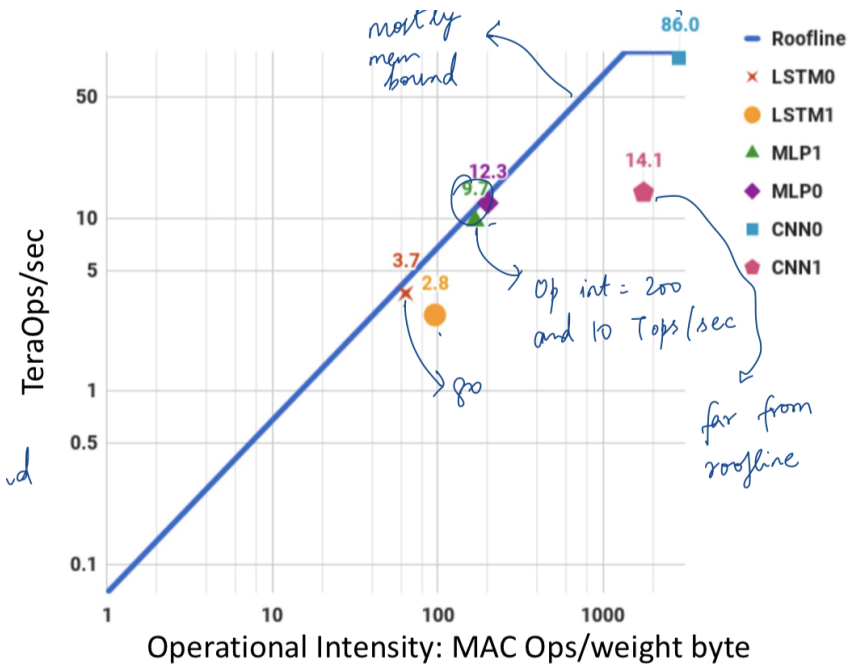


- Most area is taken by the matrix multiply units
- Model parameters are stored in on-chip DRAM cache
- Local activations are stored in SRAM unified buffer
- External interface is relatively low-bandwidth PCIe

Software execution atop TPUs:

- Uses CISC format instructions, e.g., `Read_Weights` and `MatrixMultiply`; all model layers are translated into these instructions
    - Very specific to ML inference workload
- *Systolic execution*:
    - Reading large SRAM uses much more power than doing arithmetic
    - Data streams through systolic arrays to avoid materializing the results as much as possible, unlike in Von-Neumann architecture

Roofline model for measuring performance:



- Tells you how much peak performance is possible for given *operational intensity*
- For sufficiently high intensity, performance becomes flat become all compute resources are saturated