* **System parameters:**

Number of racks: 40 (4) – indexed from 0 to 39

Number of racks connected to OCS (|***N***0|): 20 (4) – indexed from 0 to 19

Number of hosts per rack: 40 (2) – hosts are indexed from 0 to 1599

Containers per host: 20 (4) – containers are indexed from 0 to 31999

Intra-rack link rate: 8 Gbps (out of 10 Gbps)

Inter-rack electrical path rate: 6 Gbps (out of 10 Gbps)

Optical path rate: 100 Gbps

Number of links between the OCS and the core EPS: 10 (2)

* **Simulation parameters:**

Advance-reservation time slot: 10 ms

AR horizon: 50k slots – 500 s

Electrical-path update interval: 1 ms

HDFS block size: if input data size <= 1TB, 128 MB; if input data size > 1TB, 1 GB

Number of map tasks – the same as the number of input blocks

Number of reduce tasks – ceil(#maps/8)

Disk space per host: 5 TB – assuming disk space is large enough

Delay scheduling threshold: 0.1 s – the time passed before a no-local container is allocated to a job

Reduce slow start: 0.05

Reduce rampup limit: 0.5 – percentage of reduce containers that are requested before all map tasks are completed

AM initialization time: 1 s – time for application master to initialize, which is the time for an AM to first request containers after a container is allocated to the AM

Map/reduce task execution time: 15 s/ 25 s if block size is 128MB, 15\*7 s/ 25\*7 s if block size is 1GB

Node Manager (NM) heartbeat interval: 0.5 s – each host has a NM that sends heartbeats to the RM periodically to inform the RM of idle containers.

* **Assumptions**
* Disk space is large enough so that all the blocks of any dataset could fit in one rack
* Intra-rack disk-I/O access rate is equal to local disk-I/O access rate
* Each map task generates the same amount of shuffle data for each reduce task, i.e., if the total shuffle size is x, and the number of map and reduce tasks are n and m, then the size of data shuffled from any map to any reduce is x/(n\*m).
* If a job has zero shuffle data, then it is a map-only job.
* **HDFS block placement:**

Shuffle-heavy job (SHJ) definition: shuffle size is larger than 2 GB, input size/shuffle size < 1000

The second condition is chosen because there are a few jobs with very large input size say but relatively small shuffle size (e.g., job8032 has 3.8TB input data but only 2GB shuffle data). If these jobs are considered as SHJ and their input data are stored accordingly, all the map tasks need to be scheduled onto a single rack to ensure that the map output generated on that rack is large enough (>1.2GB) to justify the setup time of circuits. In this case, the execution of map tasks will be delayed due to the limited containers available in a single rack. Therefore, for those jobs violating the second condition, we will treat them as regular jobs (RJ).

We store input datasets into HDFS in the same order as job arrivals. Each input dataset corresponds to one “inputPath” given in the Facebook workload. Any job having the same input path uses the same dataset. If there is at least one job using an input path is a SHJ (based on the definition above), then the dataset of the input path is a shuffle-heavy dataset. Otherwise, the dataset is a regular one.

For each dataset, if it is a shuffle-heavy dataset, then 3 racks with largest free disk space in ***N***0 are selected, each of which stores one replica of the dataset. All blocks of the dataset are evenly stored among all the hosts in a rack. If a dataset is a regular one, then for each block, the host storing the first replica is selected randomly from the cluster. The host storing the second replica is selected randomly from all the hosts that on a different rack from the first replica, and the third replica is stored on the same rack as the second replica. All the datasets are stored into HDFS before the simulation starts, and the metadata of stored datasets are kept in a file, which can be read from when running the simulation.

* **Simulation flow**

Initializing all components including RM, NMs, SDN controller, ePath simulator and job generator.

RM is responsible for initializing AMs, handling container requests sent by AMs, maintaining rack queues and the cluster queue, and allocating containers to jobs based on requests and container status (sent by NMs). NMs inform the RM of container availability on each host by periodically sending heartbeats the RM. The SDN controller is operated in AR mode, which maintains circuit reservation status and schedules circuits based on requests sent from SHJ AMs. The ePath simulator simulates flows that are carried on the electrical paths. The job generator inserts jobs into the simulator one by one according to job arrival times given in the Facebook workload.

Every time when the simulation time reaches the arrival time of one job, the job generator will inform the RM about the new job, which will in turn initialize the job and try to allocate a container to its AM. A Job is initialized with the number of its map and reduce tasks and block locations, and resource requests for map containers. After an AM container is allocated, the job sends map-container requests the RM. For requests from a regular job, the RM puts them in the cluster queue. For requests from a shuffle-heavy job, the RM first selects map racks based on rack queue lengths and input block locations, then decides how many map containers are requested on each of the map racks, and puts them in corresponding rack queues. The RM also informs the SHJ AM of the indices of the selected map racks and the number of map containers requested per rack.

Every time when the RM receives a heartbeat from one of the NMs, if there are any idle containers, the RM will try to allocate them to jobs based on the container requests in the rack queues and the cluster queue. After allocating a container to a job, the RM informs the corresponding AM about the allocation. Then the AM assigns the container to a specific map or reduce task, and initializes the task. A task starts executing only after its input data is moved local through either the electrical paths or optical paths. For a regular-job AM, upon finishing assigning a container it also needs to recalculate its container requests and sends them to RM. For a shuffle-heavy job, when all the map tasks are completed on one of its map rack for the first time, its AM queries the RM of reduce racks. The reduce container requests will be enqueued into the reduce rack queues and marked as “waiting”. Once the AM knows which racks to use for running reduce tasks, it requests optical circuits between the first completed map rack and each of the reduce racks from the SDN controller. When another map rack finishes all the map tasks running on it, the AM requests circuits for it again. Once map output from all the map racks are moved into one reduce rack, the AM informs the RM so that the corresponding reduce container requests can be set to “ready” and start receiving container allocations.

* **Container request calculation**
* SHJ: AM only needs to inform the RM of the number of requested map and reduce containers, and the indices of racks storing the input dataset.
* RJ: for map tasks, container requests include the number of containers requested per rack and the total number of requested containers. Since we assume that the intra-rack disk-I/O access rate is equal to local disk-I/O rate, we do not distinguish between node-local and rack-local containers. For reduce tasks, container requests only include the total number of requested containers.

Assuming that we have three racks (r0, r1, r2), each of which has two hosts (h0+h1, h2+h3, h4+h5).

A regular job has three map tasks and two reduce tasks. The three input blocks are stored as follows:

replica 1 replica 2 replica 3

block 0 (map0) h1 h4 h5

block 1 (map1) h2 h0 h1

block 2 (map2) h0 h2 h3

Then the resource requests for map containers are {r0:3, r1:2, r2:1, \*:3}. After the RM allocates a container on h2 (r1) to the job, the requests kept by the RM are updated to {r0:3, r1:1, r2:1, \*:2}. Once receiving the allocated container, the AM assigns it to map task 1, updates the map container requests to {r0:2, r1:1, r2:1, \*:1} and sends them back to the RM. When more than 5% map tasks are completed. the AM requests for no more than 50% of the total reduce containers (specified by the reduce-rampup-limit parameter). This means that in this example, when one map task is completed, the AM requests one reduce container. After all the map tasks are completed, the AM requests for another reduce container.

* **Container scheduling (RM)**
* Map-rack selection for SHJ: the RM receives map container requests from a SHJ AM with the information of the total number of requested map containers, *Tm*, and the set of racks storing input dataset, ***Z***, and decides the map racks and how many containers are requested in each map rack. The shuffle data generated on each map rack should be large enough say 1.2 GB to justify the setup time (10 ms) of optical circuits. So the number of map racks *Nm* is the minimum of (shuffleSize/1.2GB) and 3 (each of the 3 racks stores one replica). After deciding the number of map racks, we choose *Nm* racks with the shortest rack-queue lengths from ***Z*** as map racks ***R****m*.

Then we need to decide how many map containers should be enqueued in each rack queue. Since the job completion time is largely determined by the completion time of the last map task, in an ideal case, we expect that map tasks on all the map racks can finish at the same time. Therefore, we distribute the *Tm* requested containers for map tasks in a way that the rack-queue lengths of all map racks are the same after enqueuing the new requests.

* Reduce-rack selection for SHJ: the reduce container requests sent by a SHJ AM only include the total number of requested reduce containers, *Tr*. First, the RM decides the number of reduce racks, *Nr*, based on the number of reduce tasks: 1 reduce rack if *Tr* ≤400; 2 reduce racks if 400≤*Tr*≤800; 3 reduce racks if *Tr*>800. Second, the RM chooses *Nr* racks with the shortest rack-queue lengths as reduce racks. Finally, the RM enqueues *Tr* requested containers for reduce tasks following the same rule as enqueuing map-container requests.
* Container allocation: when the RM receives a heartbeat from a NM about idle containers, the RM first allocates them to requests in rack queues and then, if there are any idle containers left, allocates to requests in the cluster queue. When trying to allocate a container using rack queues, the RM first finds the rack to which the container belongs, and then allocates the container to the SHJ whose request is at the top of that rack queue. When allocating containers using the cluster queue, delay scheduling is used. All the jobs in the cluster queue are sorted based on their current usage of containers in an ascending order, and the RM check jobs in the queue one by one in order until finding one suitable to allocate the container. For the current checked job, if it requests a reduce container, then the container is allocated to the current job. Otherwise, if the job requests a map container on the same rack as the idle container is, then the container is allocated to the job. In addition, if the job requests a map container on a different rack as the idle container, but the current time minus the last time the job receives a container allocation is longer than a threshold, then the container is still allocated to the job. If none of the above three conditions is satisfied, the RM continues to check the next job.
* **Container assignment (AMs)**
* Reduce container assignment: if an allocated container is for a reduce task, the AM assigns the container to the reduce task with the smallest task index, which has no been assigned a container.
* Map container assignment: if an allocated container is for a map task, the AM assigns the container to a map task based on map input block locations. First, the AM checks if there is a map task whose input block is in the same host as the container. If yes, the container is assigned to that map. Otherwise, the AM tries to find a map task whose input block is in the same rack as the container. If neither a node-local nor a rack-local map task exists, the AM assigns the container to a map task that waits for a container.
* **Task input data movement**

After a task is assigned a container, before executing task functions, it first needs to move its input data to the same host as it resides, no matter whether it is a map task or a reduce task of either a regular job or a shuffle-heavy job. For a regular job, both the map input data movement and the intermediate data shuffling is through the electrical paths. For a SHJ, the inter-rack intermediate data shuffling uses optical circuits, while the map input data movement and the reduce input data movement use electrical paths (because shuffle phase is decouple from reduce tasks for a SHJ).

* Every time when a flow of input-data movement or data shuffling starts, if the data is moved from a source rack *rs* to a different destination rack *rd*, i.e., *rs* ≠ *rd*, the number of flows on the uplink of rack *rs* is increased by 1, do does the number of flows on the downlink of rack *rd*. The ePath simulator is responsible for simulating when a flow is completed.
* For a regular job, the output of each map task needs to be shuffled to all the reduce tasks. However, the shuffling from multiple map tasks to multiple reduce tasks does not happen at the same time. Because they do not finish nor start at the same time. For example, when a map task finishes, some of the reduce tasks have been allocated containers while some have not. One the other hand, when one reduce task is assigned a container, some of the map tasks have completed while some have not. So we need to start shuffling flows based on the status of each map-reduce pair. Since shuffling from a map task to a reduce task can only start after the map task finishes and a container is assigned to the reduce task, we starts a shuffling flow following this rule.
* **Electrical paths (ePath simulator)**

The ePath simulator keeps track of the number of flows on the uplink and downlink of each rack. Periodically (every 1 ms), the simulator calculates the rates on each inter-rack link, (*Rr* for uplink and *R’r* for downlink) by diving the electrical path link rate by the number of flows on that link. Then the rate of each inter-rack flow from source rack *rs* to destination rack *rd* is updated to min{*Rrs*, *R’rd*}, and the size of data left to transmit of a flow is reduced by min{*Rrs*, *R’rd*}x1ms. In this way, a flow is completed when its size of data to transmit reaches zero.

* **Optical circuits + AR (SDN controller)**

The SDN controller is responsible for operating optical circuits in advanced-reservation mode. It maintains the availability of each optical link in each timeslot within the reservation window.

The controller receives circuit requests from AMs, which specifies the source and destination of shuffling, and the size of data to be transferred on a circuit in terms of the number of reservation slots needed. The shuffling source is a single map rack and the destination could be one or multiple reduce racks.

* If the storage is involved in shuffling data, then the time when a piece of data is pulled down by a reduce rack must be at least one slot later than the time when it is pushed into the storage by the map rack.
* Single reduce-rack shuffling: two choices of how to shuffle the data. The first one is to set up a circuit between the map rack and the reduce rack directly through the OCS. The second one is to first push the shuffle data to the storage, and then to pull it down when the circuit of the reduce rack becomes available afterwards. We choose the one with earlier shuffle completion time.
* Multiple reduce-rack shuffling: there are still two choices in this case. The first one is to shuffle the data from the map rack to all the reduce racks at the same time through multicasting, while do not need to use the storage. The second one is to first push the shuffle data to the storage, at the same of which if any of the reduce racks has an available circuit, they can get the data at the meantime. For the rest reduce racks, they can pull down the data from the storage when their circuits become available.