SCALABLE ALGORITHM DESIGN

# MAPREDUCE

## Definition

* Programming model: functional + parallel running
* An execution framework: large-scale, cluster of commodity hardware
* Batch processing – involve “full scan” data

## KEY PRINCIPLE

### Scale out - not up!

* large number of **commodity >** small number of **high-end servers**
  + I/O is slow, data process is quick
  + **share nothing** **(better)** : independent entities, no common state
  + sharing problem: synchronisation, **finite bandwidth to access data from SAN** (storage is remote of the servers)**,** different OS, not compatible

### Failure

* **normal,** due to scale and shared environment
  + hard ware/software/elecrical/cooling…

### Moving process to data

* data intensive workloads (network problem + large data) => **data locality principle**
* Distribute filesystem is needed
  + HDFS: master/slave architecture, NameNode (master) control/distribute task to DataNode (clients)
  + idea: **moving computation is cheaper than moving data:** when slave is available then NameNode will assign map-task whose input data is store on that slave (node). Otherwise, Hadoop find a task can achieve at rack-level.
    - —— Rack is a collection of many nodes that **phisically** stored close together and same network switch (bandwidth nodes in rack > between rack). ——

### Hide system-level detail

* abstract “distributed” part
* BUT, custom data reader/writer/partitioning and memory utilization
* parallel problem: because of sharing nothing => need to define problem can run parallel or not

## Function programming model

* Key: **higher order function**s (function that accept other function as argument) MAP and FOLD
  + MAP: **transformation -** happen isolation => paralleled
  + FOLD: **aggregation -** element must be brought together - we can **group =>**  run parallel
  + Associative and commutative operations: gain performance through local aggregation and re-ordering
* Data Structure: **key-value pair.** They can primitive (int, float) or arbitrary complex structure (list, tuple)
  + map (k\_1,v\_1) -> [(k\_2,v\_2)]
  + reduce (k2,[v\_2]) -> [(k\_3,v\_3)]

### Generic MapREDUCE Algorithm

* dataset stored on an underlying **distributed filesystem**, split into a number of **blocks**
* mapper is applied to input => **intermediate** key-value pair
* implicit between map-reduce is **parallel “group by”** operation on intermediate key
* reducer is applied to all **value** same **intermediate key** => output key-value pair => written back to distributed filesystem or input of subsequent MapReduce
* Combiner: reduce amount of intermediate data (mini-reducer) in mapper – CAUTION if some cases ( Ex Average)

# Basic Design Pattern

* Data structure is important - optimisation is hard
* designer not control:
  + where MAPPER&REDUCER run
  + when they finish or start
  + which input are processed by specific mapper
  + which intermediate key-value is processed by specific reducer
* designer control:
  + data structure (key-value pairs)
  + sort order of intermediate key (order in which reducer will encounter)
  + partitioning of key space (set of keys that will be countered by a particular reducer)

## Local Aggregation:

* data transfer over the network problem => **reduce intermediate data** (Combiner and preserve state across input)

### Approach1 : In-map combiner:

* Provide control over when local aggregation occurs and determine how exactly aggregation is done
  + Efficiency: reduce key-pair value across over n docs (normally, combiner only works on 1 docs)
  + Deal with reduce stragglers - frequent occurring keys (ex: “the”)
* Precautions:
  + breaks the functional programming paradigm because of **state preservation (s**ome problems depend on the order in which input key-value pairs are encountered)
  + memory capacity is limited: in-memory combining **strictly** depend on memory to store intermediate results.
* **Solution: “block”** key-value pairs and **“flush”** intermediate-key periodically (after we have n unique key-value pair)

### Pairs and Stripes: (example of word-co-occurence problem)

* Problem: large corpora => memory cannot fit the matrix (save into hard disk if work with 1 machine => scalability limitation)

|  |  |  |
| --- | --- | --- |
|  | **Pair** | **Stripe** |
| Process | * Input: key-value: offset of line – content * Mapper: emit key-value: words pair as key - integer 1 (the count) * Reducer: receive key-value at same word and compute abs count of joint event and emit find key-value value | * Input: key-value: offset of line – content * Mapper: emit key-value: word as key - associative array ( co-occurrence word and the count) as value * Reducer: receive associative array at same word. Emit key-value in form (word, associative array) |
| Comparison | * Generate large number of key-value pairs * less benefit from combiner ( <pair,1> and send through network) * does not suffer from memory paging problems | * More compact (associative array | occurrence-word and count) * Value-part is complex for serialisation * greatly benefits from combiner * suffer from paging problems |

## EXAMPLE OF RELATIVE FREQUENCY (find f(w\_j|w\_i))

* stripes approach easily work
* pairs approaches
  + mapper: emit a special key-value pair to capture the marginal (**reducer cannot hold all pairs to count)**
  + Define a custom partitioner for routing intermediate key-value pairs. Default partitioner based on hash value of intermediate key, complex key is represented by bit => same **left-key** may not meet each other => partition based on hash of left word only
  + Preserve state across multiple keys in the reducer
* **Advantage:** 
  + memory requirement is minimal: only “special key” is store
  + no buffering of individual

Hadoop Internals

# TERMINOLOGY

## MapReduce:

* **Job:** execution of a Mapper and Reducer across dataset
* **Task**: same jobs but on piece of data
* **Task Attempt**: instance of an attempt to execute a task

+ Task have many task attempts

+ same input cause crash => new attempt

## Collocate data and computation

**-** data >> => computing >> => link between computer nodes and the storage nodes becomes a bottleneck

- idea: abandon a separation between **computation and storage (**avoid moving data)

# Hadoop Distributed Filesystem (write-once read-many-times)

- Not handle **concurrency,** allow **replication**

**-** optimized for **throughput** (amount of time to complete work) **not latency** (time data travel from one to another)

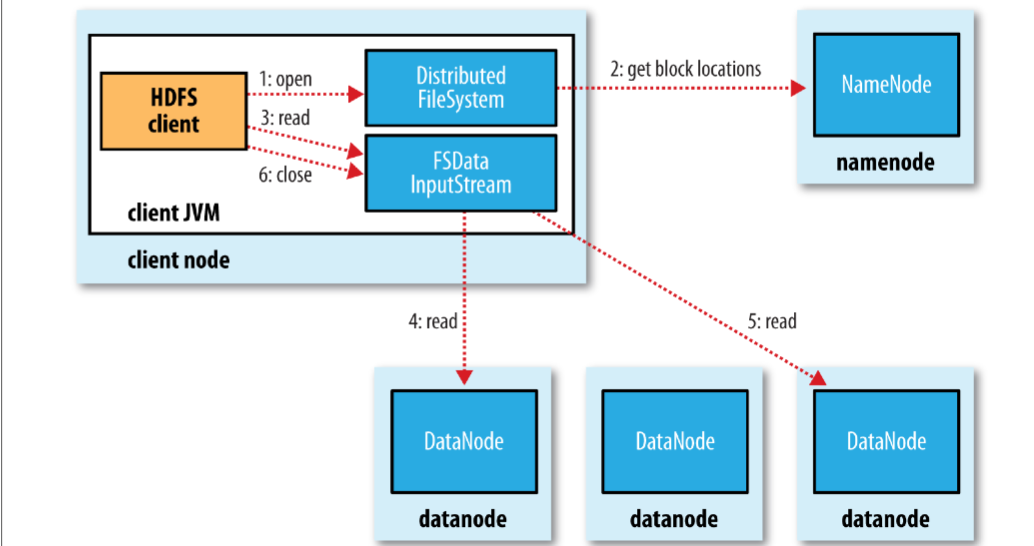
## BLOCK

* files is broken into block-side chunks
  + blocks are big => minimize the cost of seek time (seek time = 1% transfer => file > 100mb)
  + simplify storage subsystem rather than a file (fixed size - disk management and not meta-data concern)
  + replicate across another nodes

## Namenodes and data notes

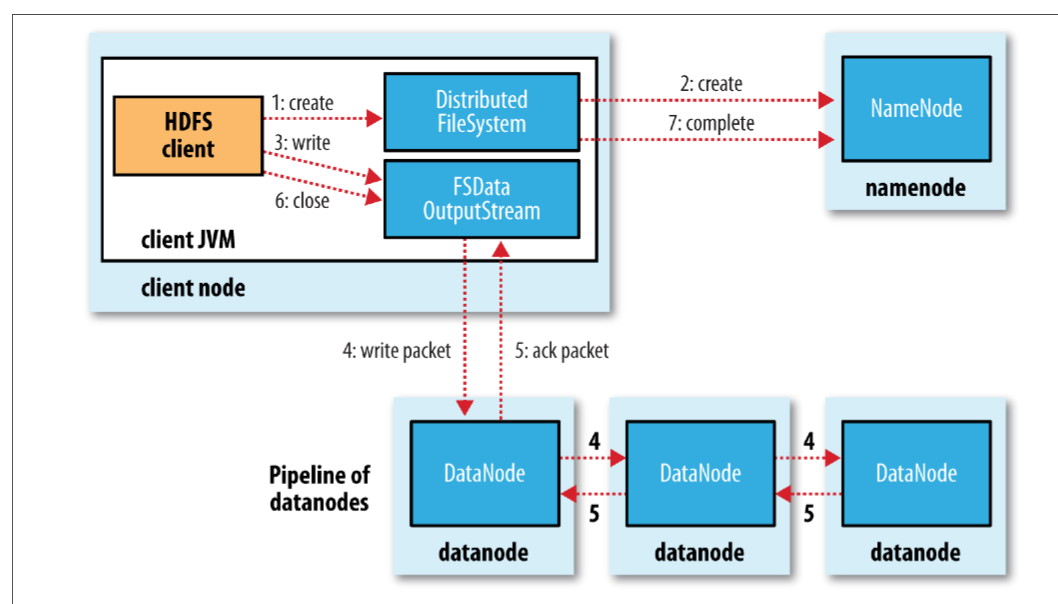
* Namenode: keep metadata - filesystem tree in **RAM**
  + not information of blocks: Datanode take care it
  + maintains over health of system
* Datanode:
  + store and retrieve data to clients
  + report periodically to Namenode the list of blocks they hold
* failure of namenode:
  + save persistent state to multiple filesystem => local disk or NFS mount
  + secondary namenode (different machine): merge namespace with edit log
  + **usual approach,** use the NFS copy of metadata and switch 2nd to primary

## Anatomy of a file read



* Namenode only used to get block location
* For each block, Namenode return set of Datanodes hold a copy
* Datanote are sorted by **their proximity to client**
  + **Network topology** (arrangement of network): sum of distance to ancestor (node -> rack -> data centre -> different data centre)

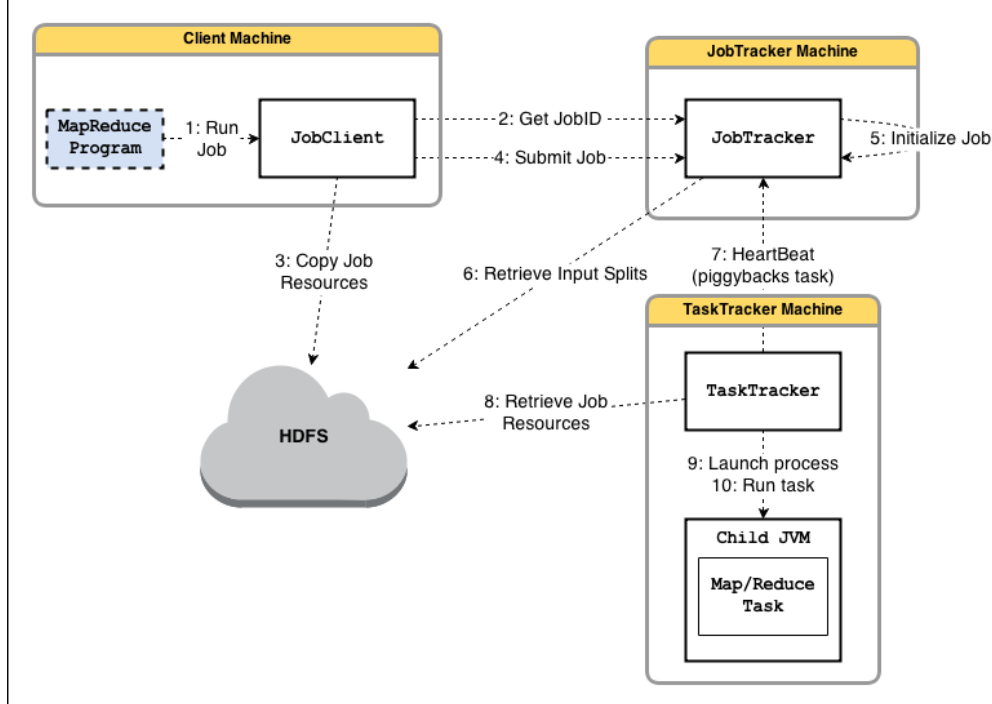
## Anatomy of file write



* Client ask Namenode for a list of suitable Datanode
* A list form pipeline: first DataNote store copy and forward it to 2nd and so on
* **Replica Replacement:** tradeoff between reliability and bandwidth
  + 1st: same node, 2nd: off-rack, 3rd: same rack but different node
* **Coherency Model:**  visibility of reads and writes for a file.
  + Block content not be visible after a write is finished (data store on data note’s memory not disk)
  + Application design: sync() to force synchronisation but tradeoff robustness/consistency and throughput

# HADOOP MAPREDUCE

## Anatomy of Mapreduce Job Run

* **Job submission (step1-4):**
  + *runJob()* method creates new instance of JobClient and call *submitJob()*
    - verification: **output specification** (output directory has not specified or system already have it) or **input split** can’t be computed => error and cannot run
  + Copy Jar of the job and replicate, there are lots of copies across the cluster => node can find an executable file from many source
* **Job Initialzation (step 5-6)**
  + Jobtracker: create an object for job, encapsulate its task and **bookkeeping** task progress and status
  + **Create list task to run:** retrieve input split (JobClient) and divided into map task**s** for each split. read configuration file to set number of reducers

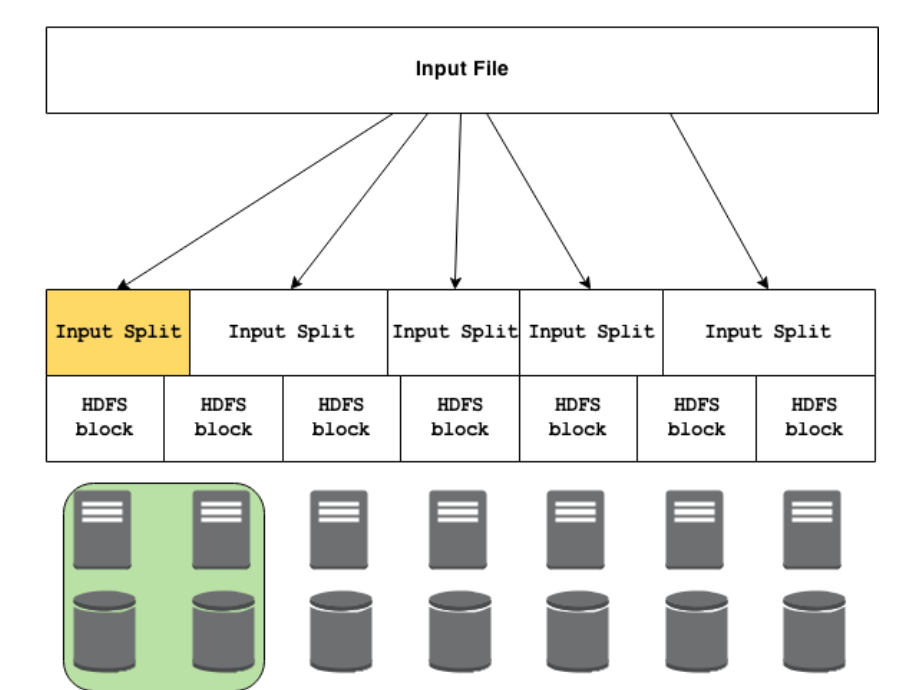
### Scheduling

* Task assignment - Heartbeat-base mechanism
  + TaskTrackers periodically send heartbeats to Jobtracker => alive or inform that it can execute a task
  + Jobtracker piggyback a task if Tasktracker available
  + Select a task:
    - JobTracker choose a job (job scheduling)
    - TaskTracker has fixed slotfor mappers and reducers
    - JobTracker give **priority to map tasks** (network- pick a task whose input split as close as possible to tasktracker that contains data note in local) - **unused for reduce tasks** (data need to shuffle and “group” before computation).
* **Task Execution**
  + Execute: copy jar from HDFS, create local directory, create instance TaskRunner
  + TaskRunner launches a child JVM:
    - Error not affect TaskTracker
    - new JVM created per *InputSplit:* use JVM reuse option (run task share JVM subsequently instead of parallel) to gain performance **(long initialization process)**
  + **Streaming and Pipes:** run and communicate with user-defined executable (map and reduce) (other languages, custom output)
* **JobScheduling Methodology**
  + **FIFO Scheduler**: long-running low priority task monopolise the cluster (block high-priority task)
  + **Fair Scheduler:** give user fair share of cluster capacity **over time**
    - Jobs placed into pool, one of each user. More jobs no more resource.
    - Guarantee minimum capacity per pool
  + **Capacity Scheduler**
    - **n**umber of queues (like Fair Scheduler’s pool) but have hierarchical (queue may be a child of another queue)
    - FIFO in each queue (scheduling with priority)
    - **excess capacity no being used by others.**

### Failure:

* Task failure
  + Map or reduce task throw a runtime exception
    - child JVM reports back to parent
    - Tasktracker logs error and mark *TaskAttempt* fail
    - Tasktracker free up a slot for another task
  + Hanging task:
    - Tasktracker not receive update (timeout)
    - Tasktracker kill child JVM
  + JobTracker will be notified of fail task
    - Avoid rescheduling on same TaskTracker. Task fails 4 times => job fail
* **Tasktracker failure**
  + Type: crash, running very slowly => heartbeat not send to JobTracker
  + JobTracker wait for a timeout => remove TaskTracker from scheduling (blacklist that TaskTracker if many fail)
  + JobTracker need **reschedule** a job and even completed task (intermediate output in failed node cannot be accessed)
* **JobTracker failure**
  + Hadoop now not have mechanism for this kind of failure
  + Future solution: multiple JobTracker coordinated by **ZooKeeper**

### Shuffle and Sort: guarantee input to reducer sorted by key

* Map side
  + Each time buffer is full, new spill created => spill is partitioned (corresponding to the reducers), then sort and run combiner **(compact map output => less data when written or transfer).** Finally, written on disk
  + Map output continue to be written on buffer while spilling to disk, buffer fill up while spilling, block map task
  + output file partition send to reducer over HTTP (40 thread by default)
* Reduce side
  + need inputs from many other TaskTrackers
    - map task finishes => notify parent **TaskTracker** => then notify **JobTracker**
    - thread in reducer **polls periodically** Jobtracker
    - CAUTION: Tasktracker do not delete local map output when reducer fetch them (transfer may fail)
    - Number of copy thread fetch map output in parallel
  + map output copied to memory, copy to disk if not fit
    - background thread merges all partial input to larger
    - if file is compressed => need decompression (take place in memory)
  + Sorting when all map outputs have been copied => merge phrase start

### Types and Formats

* Dataset is specified by **InputFormats**
  + InputFormats define input data (file, directory)
  + Identify a partition of data that form input split
    - Each split divided into records, the map process each record
    - splits and record are logical (not physically bound to file)
* RecordReader extract k-v record from input source

### HADOOP I/O

#### Data Integrity

* I/O operation on disks or network may corrupt data (**checksum mechanism)**
* HDFS transparently checksum all data during I/O - storage overhead 1% (create checksum every bytes (512))
  + DataNodes are in charge of checksum before storing data
  + run periodically in a background to verify all blocks ( “bit rot”)

#### Compression

* Why? Reduce storage requirement and speed up the data transfer
* Input split: use compression that support splitting (bzip2) - to be processed by separate mapper
* Recommend Format:
  + Compression support splitting
  + Spilt file into chucks and compress each chucks separately
  + Sequence File - Avro (like sequence file but can read and written by multiple language)

#### Serialization

* Transform structured object -> byte stream (transmission and storage)
* Hadoop have its own serialization format **Writable**

Spark Internals

clustering computing platform that designed to be fast and generality

# Introduction and Motivations

## Motivation

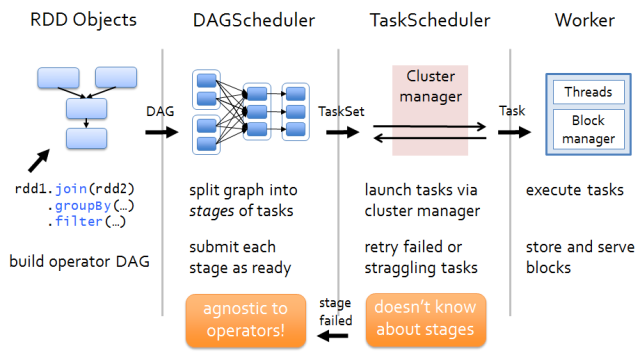
* System: unify pipeline pipeline/ simplified data flow / process speed
* data abstraction: RDD / easy to extend with new operators / descriptive computing model

|  |  |  |
| --- | --- | --- |
|  | Spark | Hadoop |
| Generality | unified pipeline | many modules for specialized jobs |
| Simplified data flow | read in ETL, store in RDD during process (ETL-train-query) then write after finish process | read – write on disk on each step (ETL - train - query) |
| Simplicity | descriptive computing model | boated computing (code structure) |
| Faster Processing Speed | simplified data flow, avoid to materialize data on HDFS |  |

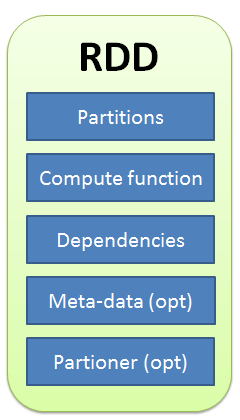
+ Transformation: user code -> distributed data in parallel (optimisation - only do transformation when call action for your algorithm)

+ Action: distributed data -> assemble output

## Anatomy of Spark Application

* The client will submit an application to SparkMaster. In there, it runs DriverProgram.
* DriverProgram starts by creating SparkContext, then it creates RDD and performs transform RDD with the filter transformation and call action count to construct DAG graph submit a job to DAGScheduler.
* In there, DAGScheduler – where optimizations happen – is responsible for converting DAG (a spark program is implicitly represented by directed acyclic graph) into**tasks**. Then mapping transformations together to merge them into a set of **stages**. Finally, it sends to cluster as ready and re-submit with failed tasks (lost output)
* TheDriver Program contacts theCluster Manager to ask for resources to launch executors, which is responsible for running the individual tasks. Once it launched, it typically runs for an entire lifetime of an application.
* The cluster manager launchesexecutors on behalf of the driver program.
* The driver process then tries to schedule each task in **appropriate executors**, based on data placement.
* Tasks are run on executor processes to compute, re-launch it several times with failed tasks and save results in in-memory or storage.

# Resilient Distributed Dataset?

* RDD is one of the new and important components in Spark:
  + **immutable objects**. It is safety to share across process with parallel processing (apply operation -> new layer of RDD)
  + **partitioned** and **distributed** across machines (scalability)
  + store **in memory** (fast) and **re-build** a lost partition (fault tolerance)
  + contains any **types of Python, Java** and even user defined class (generality)

## RDD interface

* Set of **partitions** called “splits”
* List of **dependencies** on parent RDDs (changes of transformation)
  + narrow dependency: a child is obtained from 1 parents (map, filter, union)
  + wide dependency: a child is derived from 2 or more parents (groupbykey, join)
* **Function** to compute (do action) a partition given parents
* Optional **preferred location** (meta-data)  and  **partitioning**info
* Hadoop RDD example

|  |  |  |  |
| --- | --- | --- | --- |
|  | **HadoopRDD** | **FilteredRDD** | **JoinedRDD** |
| partitions | HDFS block | same as parent RDD | one per reduce task |
| dependencies | none | one-to-one parent | shuffle on parents |
| computation | read HDFS | compute parent and filter it | read and join data |
| perferedLocation | HDFS block location | parent | none |
| partitioner | none | none | HashPartitioner |

## RDD operation: Transformations and Actions

### Transformation

* **an operation** on RDD that define how they should be transformed
* Transformation is **lazily evaluated** (only executed when action is called), allows for **optimizations** to take place before execution (combining operations before doing actions)
* return type: **RDD**

### Action

* Apply**transformation chains** (only now transformation will start) on RDD, eventually performing some additional operation
* return type: **built-in scala/ java**
* result go back driver program, storage system, cached computed RDDs somewhere to reuse it later. But when memory doesn’t fit data, spill it on disk

# Caching and Storage

* Access I/O external data source: HDFS, local disk, RAM, remote data
* Caches RDDs using a variety of storage level
  + Cache manage: use block manager to perform caching
  + Block manager: distributed key/value store
    - serve shuffle data and cached RDDs
    - track storage level for each block ( spills data to disk if memory is insufficient)
    - handle data replica

Scheduling Principle

# Objective

* large scale system is expensive, need to use them well (resource management and scheduling)
  + Cluster utilisation and efficiency are indicators
* Scalability bottleneck (cluster grow -> scheduling is more complex)
* Objective to: priority of job, per-job constraint, failure tolerance, scalability

# Current Scheduler Architecture

* Monolitic: use a centralised scheduling and resource management algorithm for all just
  + hard to add more scheduling policies and large cluster size
* 2 Level: single “resource management” that grants resources to independent “framework schedulers”
  + hadoop 1.0 vs hadoop 2.0

## Taxonomy of scheduling

### Work partition:

* Workload oblivious load-balance : chia deu
* Work partitioning and specialised schedulers: chia tuỳ theo resource needed
* Hybrid

### Resource choice:

* Which resource is available for concurrent framework

### Interference

* Multiple framework uses same resource
  + Pessimistic: make sure to avoid all conflicts by partitioning resource across frameworks
  + Optimistic: hope for the best, detect and undo conflict claims

### Allocation Granularity: task scheduling policies

* All-or-nothing (gang-scheduling): job cannot run until all slots are acquired
* Elastic, hoarding: job start when it allocates few slots, get more later

## Monolithic Scheduler

* Single centre instances: apply same scheduling algorithm for up-coming jobs
* Alternative design: multiple code path (different scheduling logic) for different jobs, but hard to maintain and implement

## Statically Partitioned scheduling

* Problem: fragment of resource
* Each framework have control over a set of resource
* Ex: Hadoop 1.0, Quincy

## Two-level scheduling

* Dynamic allocation resource, use “logically centralized” to control “resource grant”
* Meos: available resource is offered to competing framework
  + Avoid interference by exclusive offer
* Yarn: close to “monotonic scheduler”

# YARN

* Multiple application: separate resource scheduling from application logic
* Improved cluster utilization: sharing clusters over multiple app, genetic slots instead of fixed mapper/reducer
* Improved scalability: remove complex app logic from resource management
* App Agility:

## Architecture

### Design decision

* No static resource partition
* Separate resource management from application

### Resource manager (RM)

* Global resource manager and scheduler
* Run on master node, has “pluggable scheduler” (different scheduling algorithm to optimize schedule)
  + The Capacity scheduler
  + The Fair scheduler
  + Dominant Resource Fairness
* Operation
  + Node management: check health from Node manager
  + Contain management: AM request, dellocate when finish
  + AM management: create container for new AM

### Node manager

* Run on slave node
* Operation:
  + communicate with RM to report utilization
  + manage process of containers
  + logging service to HDFS

### Resource container

* Created by RM on request
* Allocate amount resource on slave node
* Application run in 1 or more containers

### Application master:

* One per application, negotiate with RM for resources, work with node manager to execute and monitor containers

## Fault Tolerance

* Container failure: AM reattempt that container again, man re-attempt => fail job
* AM failure: RM re-attempt whole application
  + Job recovery, use state to find success container and fail container
* NM failure: stop send heartbeat, RM remove it from active list
  + AM on failed node will re-submited completely