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)
    - Name node contains block information
  + 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 (**speculative execution)**

+ 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)

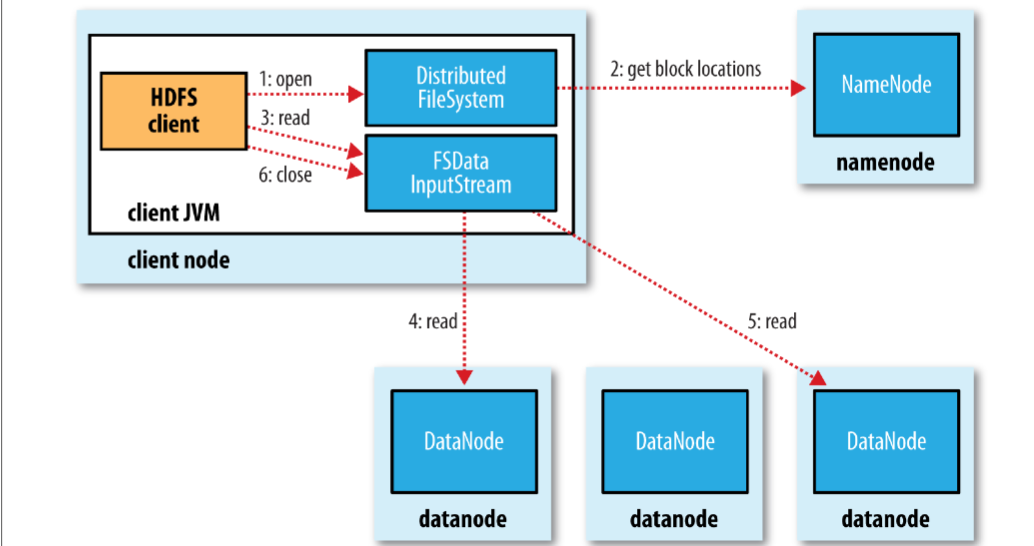
## BLOCK

* 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 (chain replication – handled by storage node themselves)

## Namenodes and data notes

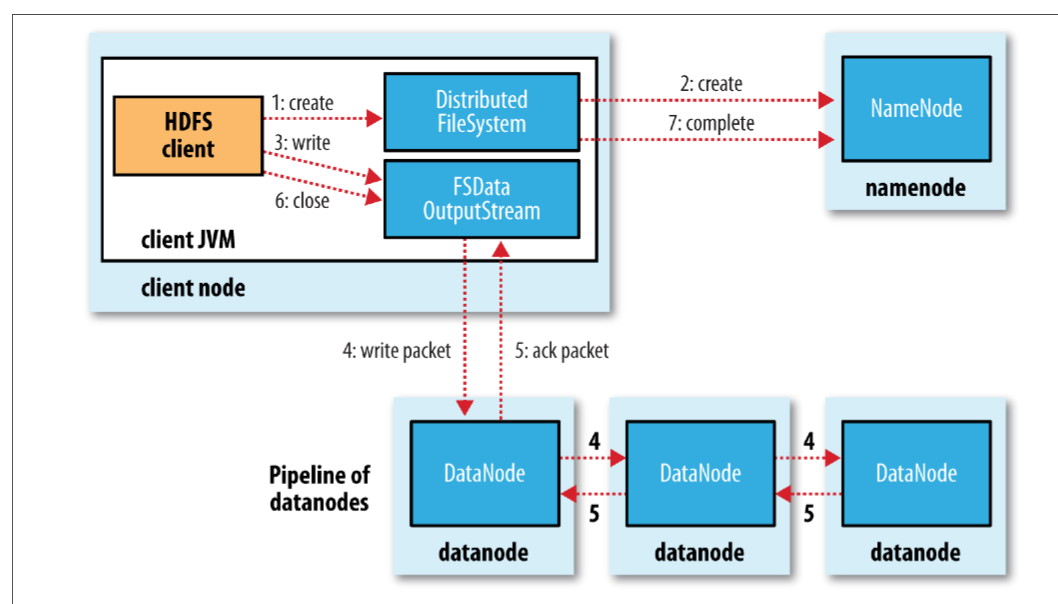
* Namenode: keep metadata - filesystem tree in **RAM**
  + not information of blocks: Datanode take care it
  + maintains over health of system
* Datanode:
  + store and retrieve data to clients
  + report periodically to Namenode the list of blocks they hold
* failure of namenode:
  + save persistent 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

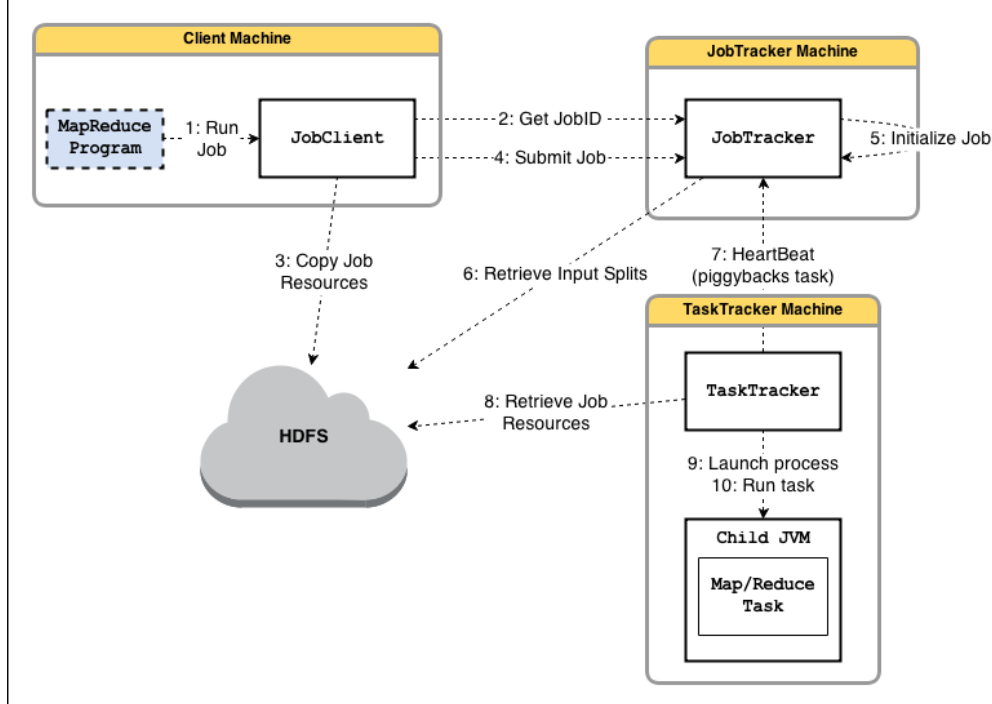


* Client ask Namenode for a list of suitable Datanode
* A list form pipeline: first DataNote store copy and forward it to 2nd and so on
* **Replica Replacement:** tradeoff between reliability and bandwidth
  + 1st: same node, 2nd: off-rack, 3rd: same rack but different node
* **Coherency Model:**  visibility 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

https://docs.google.com/file/d/0B-zw6KHOtbT4MmRkZWJjYzEtYjI3Ni00NTFjLWE0OGItYTU5OGMxYjc0N2M1/edit?pli=1

# 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

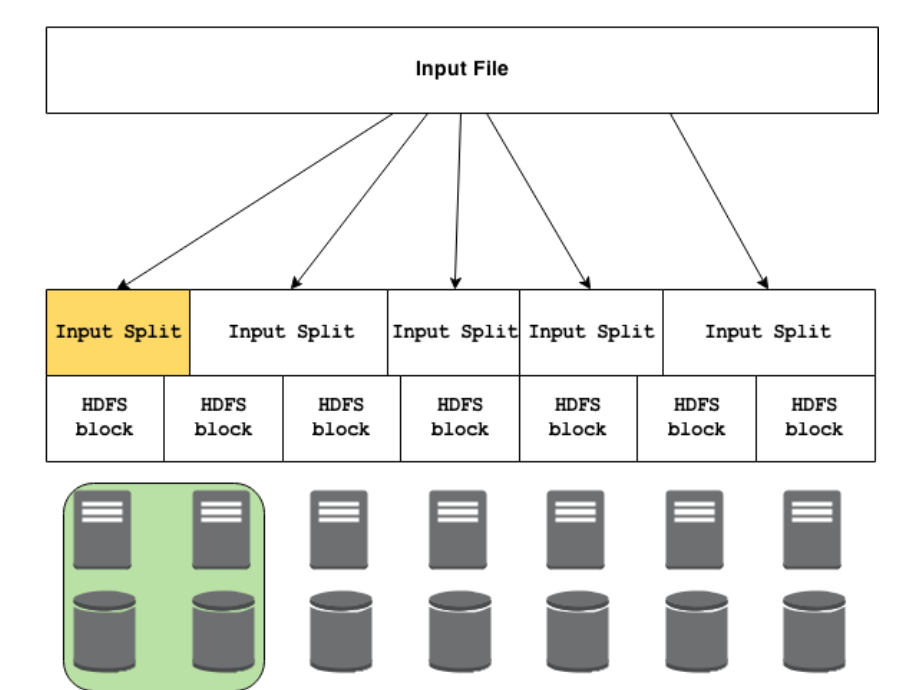
### Scheduling

* 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 (not useful) 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

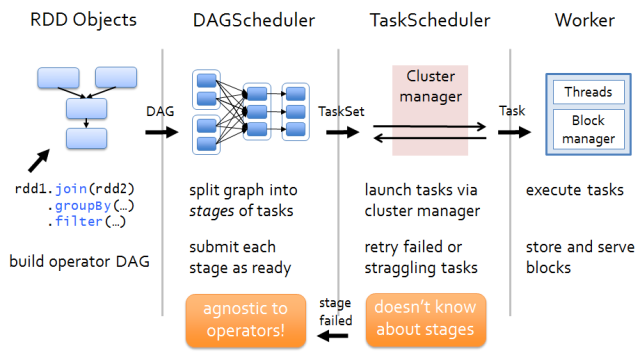
* System: unify pipeline pipeline/ simplified data flow / process speed
* data abstraction: RDD / easy to extend with new operators / descriptive computing model

|  |  |  |
| --- | --- | --- |
|  | 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 |  |

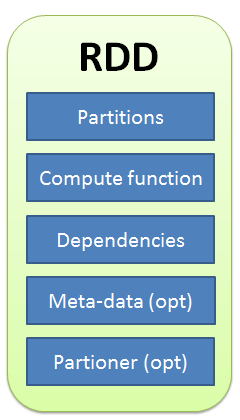
+ Transformation: user code -> distributed data in parallel (optimisation - only do transformation when call action for your algorithm)

+ 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

# Worker – Excecutor

* Host has many worker, worker have 1 JVM, Each worker spawn many executors, Executor have many threads that uses to run Task ( task can run in one or more thread). Excecutor process statically even with no threads. => faster than MAPREDUCE, JVM established again for every new task.

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. Single process/ sequential
* 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 (generic resource brokering from app logic), genetic slots instead of fixed mapper/reducer
* Improved scalability: remove complex app logic from resource management
* App Agility: multiple ver of application, easily upgrade framework

## 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 algorithms 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

# Mesos

## Motivation

* Multiple cluster computing framework share same clusters (avoid statistical multiplexing [bandwidth] – share dataset [replication is expensive] )
* Current framework (Hadoop MapReduce, Apache Spark, Microsoft Dryad) already implement fine-grained sharing but only tasks in the job level => mesos performs fine-grained sharing across different computing framework
* Problems of current approaches: mismatch different allocation granularities, no mechanism for shorted-live task
* Hypothesis: short-task, resource free quickly, allow data locality

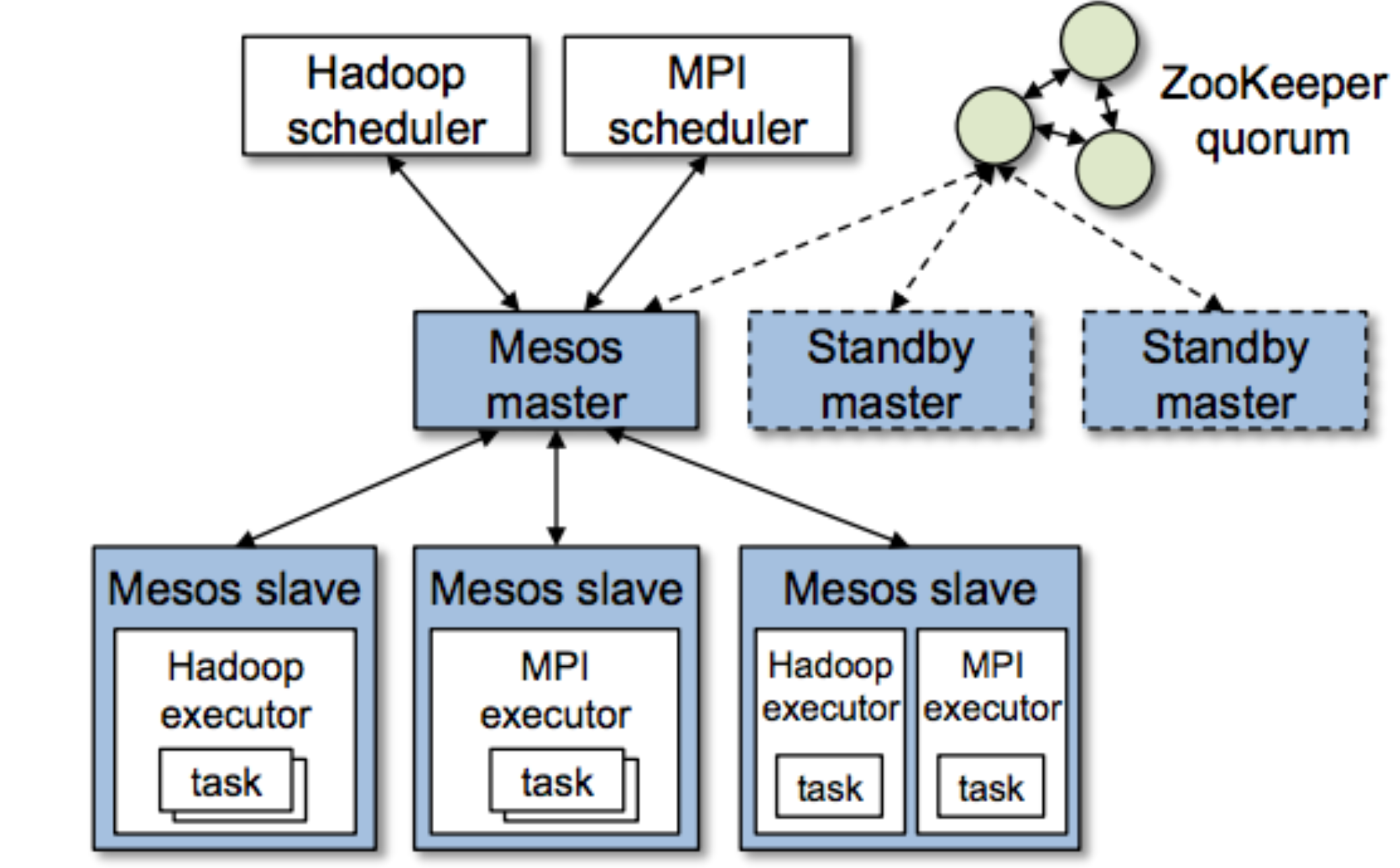
## Key design

* Input: framework requirement, instantaneous resource availability, organization policies
* Output: global schedule for all tasks of all jobs of all frameworks
* NO centralized approach: complexity, scalability, resilience, moving framework scheduling to centralized is more expensive
* Decentralized approach: based on resource offer, offer resource to framework, framework decide to use those resource (accept and organize)
* Workload: data warehouse (heterogeneous map-reduce, ad-hoc query, large scale ML, SQL-like query)

## Design philosophy

* Data centre OS: scalable an resilient core (low-level interface) common functionalities (high-level librairies)
* Resource sharing: Mesos manage resources, Framework control task and scheduleing
* Two-level approach: Framework (independent): support diverse scheduling requirement. Mesos keeps simple, minimize the rate of changes

## Architecture

* Mesos master: use **Resource offer** to implement fine-grained sharing, collect resource utilitization from slaves => offer lists of resource to slave (first scheduling – policies: fair sharing, priority based)
* Mesos framework:
  + Framework scheduler: register to master, select resource offer, schedule task based on that
  + Frame executor: take care framework task
  + **This is second scheduling:** one framework per app, framework decided how to work with tasks, jobs
* Consequence: no exposed framework requirement to master, offers is not satisfied with framework => reject and wait for new offer. Mater can implement **filter** to optimize resource offer

### Resource allocation

* Pluggable allocation: max-min fairness, strict priority
* Pre-emption primitive: have knowledge about potential resource by framework
* Guaranteed allocation: minimum set of resource by framework (lower -> never kill, higher -> kill any task)
* Isolation (multiple tasks (executors) in same node): Containers and Solaris Cages

### Scalability

* Filter mechanism: short-circuit the rejection process, avoid unnecessary communication (slave machines in network, check resource availability on slaves)
* Mesos can decided to invalidate the offer => avoid blocking and misbehaviour

### Fault tolerance

* Master designed with *SOFT STATE :*List of active slaves/registered networks/running tasks
* Multiple masters in hot-standby mode: leader selection by zookeeper
* Master reports health to framework, allows multiple scheduler in framework

## Behaviour

* Ideal: Elastic framework, support scale. Task duration is homogenous, no strict reference over cluster node
  + Elastic framework: use resources as soon as acquire and release them as soon as finish. (contrast with rigid framework)
  + Resource: Mandatory (resource must acquire to work, < guaranteed share), preferred (resource should acquire for better performance)
  + Task runtime distribution: homogeneous (constant) or not
* Limitation:
  + Starvation: larges job wait and not release resource because not satisfy minimum requirement
    - “minimum offer size” mechanism, not offer until can offer minimum resource to run
  + Fragmentation: heterogeneous resource demand => cannot optimize bin packing, waste space
    - Cluster with large nodes running small tasks => better ultilitization

# Borg

Jobs -> more tasks, tasks run same binary. Jobs run in set of machines managed as unit called cells

## User perspective

* Workload:
  + Long-run service ( shouldn’t be down) – handle short-lived latency sensitive resquest
  + Batch jobs (less sensitive to short-term performance fluctuation)
  + Workload in cells is dynamic (depending on tenant, varies with time)
* Cluster and cell
  + Clusters: set of machines connected by high-performance data centre
  + Machines: physical servers
* Job and task
  + Can interact with live jobs (update specifications of task)
  + “side-effect”: always restart (push new binary), might require migration (change specification), …

Relation Algebra

numbers of operations on data that works well Relational Algebra

* Relational Database Management System, query -> retrieve small data
* Map-reduce -> full scan (big data) to find useful information, query -> not selective, all data

### Terminology

* *relation:* table
* *attribute:* column header
* *schema:* set of attribute
* tuple: row

### Operators

* Relations (big) can be stored in HDFS
* **Selection:** sigma\_c(R): *get all tuples* of R that satisfy condition C
  + + selection only map task, use multiple reducers to write on different machine. Output is not needed a relation
* **Projection:** pi\_s(R): *get a relation* (subset of R) contains tuples with attributes in S
  + + same as selection, use reducer to **duplicate elimination (**associative and commutative) => can use Combiner
* **Union, Intersection, Difference:** (*set [bag] operator*) apply to 2 sets of tuples on 2 relations **same schema**
  + + Union: Map task assigned chunks from R and S, pass to reducer => duplicate elimnation
  + + Intersection: same union in mapper, reducer only emit if value [t,t]
  + + Difference R-S: Mapper, emit (t,’R’) or (t,’S’), For each key t in reducer: emit (t, t) if [‘R’,’R’] or [‘R’] and emit (t, NULL) if [‘R’,’S’],[’S’,’R’] or [’S’]
* **Natural join R x S:** compare pairs of tuples in 2 relations, common attribute is same => output tuples has attributes of 2 relations
  + + Mapper emit (b,[R,a]) and (b,[S,c]). Reducer: key b will be associated a list (‘R’,a) and (’S’,c) => emit (b,[(a\_1, b, c\_1),(a\_2, b, c\_2),…]
* **Grouping and Aggregation gamma\_x(R):** 
  + + partition R’s tuples according their value in **set of** attribute G
  + + each group, aggregate the value in certain attribute (SUM, COUNT, AVG,…).**phi(A)**
  + + Map prepare for grouping, reducer compute aggregation.

Apache Pig

# Use case:

* Roll-up (calculate total or sub-total after grouping) aggregates: successive aggregations, join followed by aggregations
  + dataset too big, data curation is too costly with OLAP
  + aggregated user activity logs, frequency of search terms aggregated over day, weeks, month
* Temporal analysis
  + study how to search query change over time
  + support operator that minimise memory footprint, RDBMS with JOINS over large dataset do not fit into memory => slow
* Session Analysis
  + study sequence of page views, clicks: click pattern variations in time
  + Pig support advanced data structure

# Pig latin

* high-level programming language, declarative query (SQL) and imperative (procedural programming Mapreduce) world
* multi-valued, data structure with powerful data transformation primitives

## Pig Execution Environments

* programmer focus on data and analysis
* **Pig complier:** operator are translated into Mapreduce code.
* **Pig optimiser:** data flows undergo an automatic optimisation phase. (rule-based only)
* Pig is not RDBMS => not suitable for all data processing task
* Designed for batch programming: compiles to Mapreduce, data is materialised as files on HDFS (don’t have **update operator** because HDFS contain immutable data)
* Not designed for random access (full-scan oriented)

## Pig vs RDBMS

* Pig Latin is similar to SQL
* Data-flow vs declarative programming language
  + data-flow: step-by-step operations, each operation is transformation
  + Declarative: set of constrains => applied input to generate output
* RDBMS: data in table (predefined and strict),
  + RDBMS need normalisation (create tables that relative by key)
  + Pig use data structures (defined in run-time for readability). UDF and steaming together with nested data structure => flexible
  + Nested data model (Map<docID,set<doc>>: more natural, data usually stored in nested, **algebraic language,** writing UDFs (set defined function)

## Data flow

* Pig automatically forms efficient pipelines out of sequences of per-record processing steps.
* Pig exploits the distributive and algebraic properties of certain aggregation functions (COUNT, SUM, AVERAGE and some userdefined functions) and automatically performs partial aggregation early (known as combining in the Map-Reduce framework), to reduce data sizes prior to the expensive data partitioning operation.

## Interoperability

* flexible : read-only workload (data), scan-only (schema) workload (not need fixed table, just read data and work on that)
* data curation: management of data through its lifecycle of interest and usefulness => maintain, retrieval data overtime) because of lot of data, lots of them used once and throw away, you cannot manage it.
* Data I/O is simplified (no curate, import, schema)

## user defined function

* SQL restriction(only scalar function used in SELECT, set-valued function used in FROM, aggregation function used in GROUPBY)
* Support multiple language

PIG LATEN

# Data model:

* Atom: atomic value as string or number
* Tuple: sequence of fields
* Bag: collection of tuples, flexible schema. Ex { (‘a’,’b’(,(‘a’,(‘b’,’c’)) }
* Map: collection of data item ( key as atom) { key -> (bag) }

# Statement:

* Execution
  + statement is parsed
  + Interpreter build logical plan for every relational operation (verify validity of inputs files and bags)
  + logical plan of each statement => add to program
  + interpreter move on to next statement
* **Lazy Evaluation: no processing during construction logical** 
  + just check syntax not data
* not start any processing until whole flow is defined (optimisation later)
* **DUMP** or **STORE** to start execution
  + logical plan is compiled into physical plan
  + prepare a series of Mapreduce JOB => sent to Hadoop cluster
  + **EXPLAIN** show MAP-REDUCE plan
  + STORE: program/job optimisation
    - Multiple query: B and C derived from A, normally, read A -> create B, read A -> create C. Now read A -> create B and C

# Schema

* **DESCRIBE:** display schema in relation
* Schema declaration: flexible but awkward in reuse ??
* Cannot constrain => null value in data => **SPLIT** to partition good and bad records

# Data processing Operator

* LOAD AND STORE
* tuple processing:
  + + FILTER <conditions>
  + + FOREACH <bag type> GENERATE <list of expressions> (apply to each tuples)
  + + STREAM using external program or script
* related data grouping:
  + + GROUP
  + + COGROUP create tuple of bag with same grouping attribute

# Execution Engine

## Build Logical plan

* Pig interpreter parses the command, verify the validity of inputs file and bags
* No process is carried out during constructing logical plan (strigger when call DUMP, STORE, but at that time logical -> physical plan)
* Lazy execution model (allow in-memory pipelining, file reordering, various optimization)

## Map-reduce plan (physical plan)

* GROUP BY can run parallel (map -> assign key for grouping, reducer -> process a group at a time)
* CO-GROUP command -> distinct map-reduce job (map-> assigned key to tuples based on “BY” clauses”, reducer -> no-operation)
* CO-GROUP with more than one input data (map -> extra field to identify which data set the tuples come from, reducer -> decode this information to insert into appropriate nested bag)
* Command after CO GROUP C\_i can put (a) into reducer C\_i or (b) into mapper C\_(i+1)
  + Choose (a) option because reducer data “materialized” between jobs
* Other paralism
  + LOAD : operating over HDFS
  + Filter, FOREACH: map-reduce
  + CO-GROUP: output of maps are repartitioned in parallel to multiple reducers
  + SORT: 2 jobs ( samples input to determine quantiles of sort key, range partition based on quantiles in map and sort on reduce pharse)

# Optimization

* Logical: query plan
  + Early projection, early filtering, operator rewrite
* Physical: execution plan
  + Splitting logical into multiple physical ones
  + Join execution strategies
* Efficiency with nested bags
  + CO GROUP place tuples in same group into nested bag => avoid materialize these bag => optimization
    - Can apply **distributive and algebraic** function
  + Algebraic function:
    - Tree with sub functions, each leaf operate over sub-set of input data
  + Some cases COGROUP still not efficient
    - Non-algebraic function => nested bag spill to disk. Pig support disk-resident bag implementation (data on disk and will be read into memory when do operation) to sort and duplicate elimination

Coordinating distributed system

# What is the distributed system

* Multi-thread process, multi-process on single server, in set of server in data centre, set of geographically distributed data centre
* Why? Resource & data sharing, Availability, Scalability, Performance (horizontal distribution: more machines do same task), Vertical distribution (multi-layers, multi-tiers architecture)
* Consensus problem (synchronous across distributed system). Some functionalities:
  + Aggregate function
  + Synchronization: agree on value
  + Reliable broadcast: message sent -> all received or none of them
  + Atomic commit: processes reach common decision (abort or commit)

# CAP theorem model

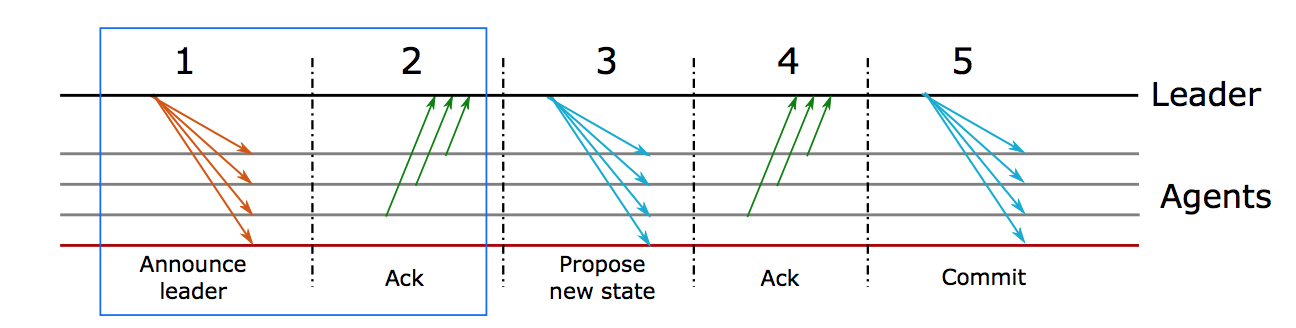
* Linearizability (CONSISTENCY): every read have to receive the most recent write or an error
* Availability: every request get answer (even not correct but no error)
* Partition tolerance: system continue working correctly even network lose or delay
* CAP theorem: only chose 2 properties over 3. Cannot dismiss Partition Tolerance (data still run) property => AP or CP, can switch to each other’s
* CP-oriented systems: BigTable, Hbase, MongoDB, Redis, MemCacheDB, Scalaris, ZooKeeper1
* AP-oriented systems: Amazon Dynamo, CouchDB, Cassandra2 , SimpleDB, Riak, Voldemort
* Consistency model:
  + Eventual consistency: after a successful write, **eventually** every read will return written value,
  + Strong consistency: after a successful write, all reads must return new value
* Fault examples:
  + Simple: network partition, hardware or software crash, outdate… or complex: feedback loops that overcompensate
* Design tolerance mechanisms:
  + Replication: load balancing, master-slave
  + Isolation: malfunctioning components not affect whole system

# Consensus protocols (Consistency in data aross over nodes)

## Two phase commit:

* P1: coordinate suggest value to other nodes and wait for response
* P2: if all nodes agree => C send commit command
* Failure: coordinate fail, nothing can be done until it restart but if one of nodes fail => slow or unreadable => value cannot be committed

## State-manchine replication

* Paxos and Raftt, it is fully deterministic
* Committed operations are executed by all state machines in the cluster in same order
* Paxos (complex and difficult to implement) round
  + P1: a note self-appoint leader and chose new ballot ID, send a ballot proposal to other, other node return highest ballot ID they know, a majority responds with the proposed ID, the protocol can process, other node can include their proposal for the value
  + P2: leader resolve any conflict on value. Leader propose new value (3) then receive answer (4). If majority accepts, it is final for this round. Otherwise, start new round.
* More complicated because still not explain any thing about state machine

# RAFT

## Introduction

* The elected leader handles all communication (no communication until leader is elected)
* Raft maintains a distributed log contain state machine command. Snapshot is used to kept a log size limited
* All nodes are known in advance – passive replicator – heartbeat mechanism

### Node state:

* Leader: handle clients requests, manage the log
* Follower: passively replicate the log and state machine
* candidate: transition state used during elections

### Election terms:

* Time is divided into term, identified by ID
* All messages labeled with term id, node record term that they believe to be current term

### Election:

* Each node has “election timeout”, if it doesn’t receive heartbeat from leader before timeout => The node start election
  + Increase termID, become CANDIDATE and vote itself
  + Send vote request to other nodes
    - Receive majority votes => become leader
    - Receive a message from leader => follower
    - Noone win before election timeout => restart
* Properties:
  + Safety: 1 leaders / term, voters vote once per term
  + Liveness: leader **eventually** will be elected. Election is started after **random timeout.** It makes sure different candidate at different time

### Normal operation:

* Clients send commands to the current leader
* The leader **logs** a new (uncommitted) entry
* The leader sends the new entry to all nodes in the next heartbeat
* Once a majority answers, the leader commits the new entry (commit means changes value in their state)
* The leader answers the client
* The leader asks all nodes to commit the entry in the next heartbeat
* The nodes commit the entry in their logs

### Failure

* If leaders fail: timeout -> election. Uncommitted entries: user doesn’t receive commit ack => retry later, uncommitted entries on follower will be overwritten by new leader
  + That leader revive: become followers, new leader will send missing log file
* If follower fail: nothing happen.
* 50/50 vote: no leader, timeout => new election
  + low probability: 2 candidate at the same time (random timeout) initialize message for new election at the same time
* Network partition: **majority rule** ensure over ½ of partition commits new entries
  + Bigger partition work normally, smaller partition cannot do any thing because of majority rule
  + Smaller partition is healed => receive heartbeat higher ID => leader of minority partition step down => update missing log

# Zookeeper

* Common service for distributed system: configuration, group management, naming, presence protocol, distributed synchronization
* Simple interface, highly available architecture

## Architecture

* ZAB consensus protocol.

|  |  |
| --- | --- |
| ZAB | PAXOS |
| Primary-backup | State-machine replication |
| Replica agree on application order of incremental **state update**, generated by primary and sent to replica. State update must be executed in exact original order of the primary. => Primary die, new primary cannot change the order   * + - * + Concurrently agree on the order of multiple update | Execute same sequence of **client requests**. These requests are summited concurrently => can be different order on replication  If leader die, new leader executes recovery and can arbitrarily change order “submitted requests”   * + - * + Some case, the order is quite important         + Workaround: sequentially update, 1 uncommitted at a time |

* User can talk with any server but all updates are handled by elected leader.
* Data kept in memory for performance. Snapshot and transaction logs on persistence storage
* AP by default => data on cached can be old. Can switch to CP (sync before read but speed penalty)

### Znode

* Regular: created and destroyed by the client
* Ephemeral: created by the client, but ZooKeeper will delete it if the client disconnects
* Sequential: created by the client, but the name is generated by ZooKeeper using a counter
* Ephemeral + Sequential: combines the two above

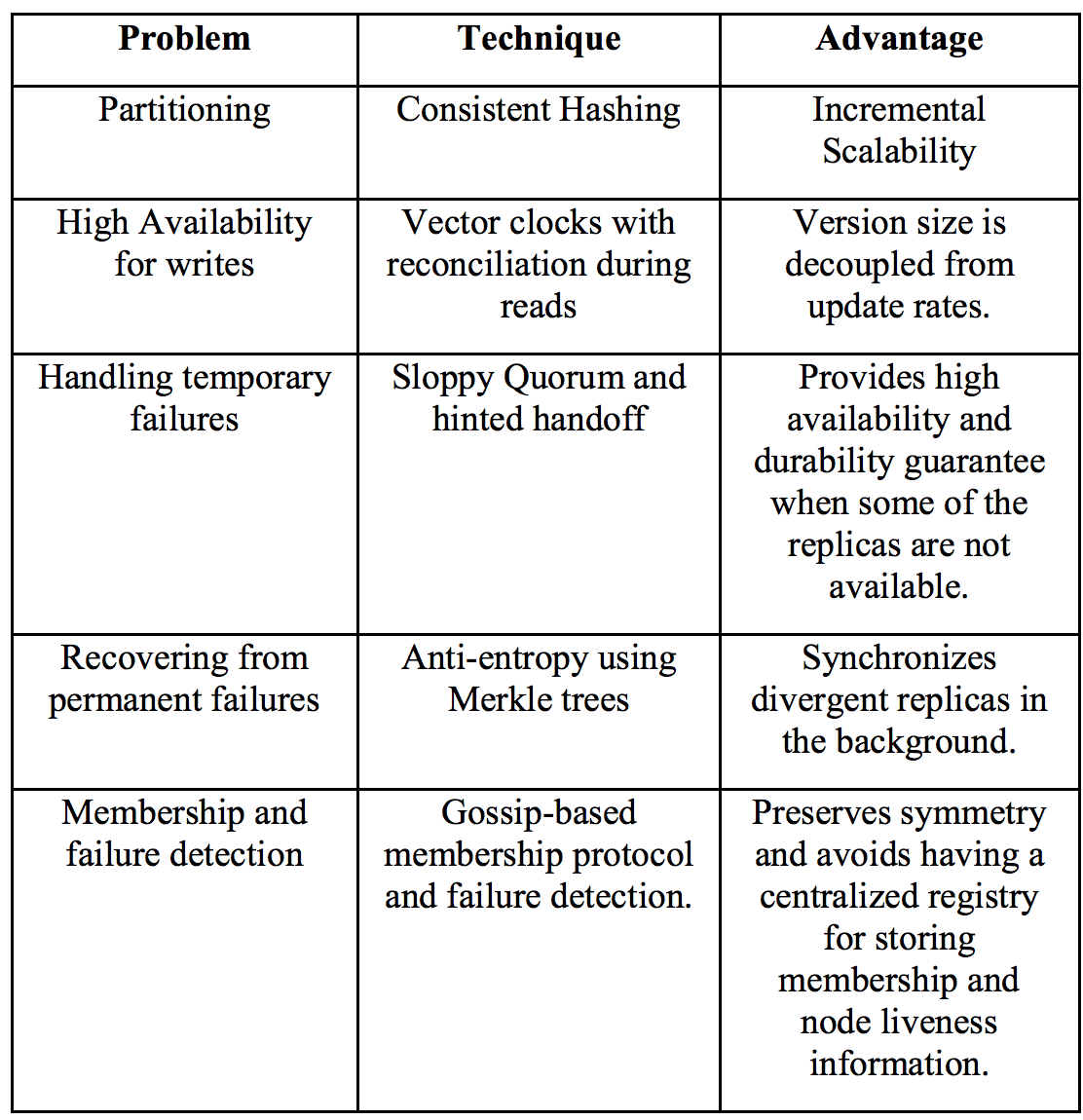
Distributed Storage System

# CAP Theorem

* CAP theorem is a fundamental theorem stating the tradeoffs among system properties
* Choose C or A that depends on app logic

# Amazon Dynamo

* Amazon web service does’t not emphasis on consistency => AP
* Properties: Basically Available – Soft state – Eventually consistent (BASE) AND ACID (Atomicity, Consistency, Isolation, Durability)
* Amazon Dynamo:
  + Work behind many context of AWS (power client facing service, internal amazon service <~ (shopping page))
  + Highly available key-value storage system (why use key-value? store small object <1MB)
  + Favours availability over consistency under failures
  + Service use Dynamo: not need transaction, only need primary-key to access
  + Properties: low latency (time interval between stimulation and response), Scalability, Always-on availability (especially for write), Fault Tolerance, Eventually consistency



## Data Partitioning

* Because of scale, require a mechanism to dynamically partition data (keys) over the set of nodes
* Consistent Hashing: Hashes of keys give m-bit identifiers, Hashes of nodes give m-bit identifiers, Identifiers are ordered in “cirle” or “ring”
  + Key is assigned to closest **successor node id** ( key k is assigned to first node which has ID >=k )
  + If such node does not exist, navigate the circles and find the smallest ID

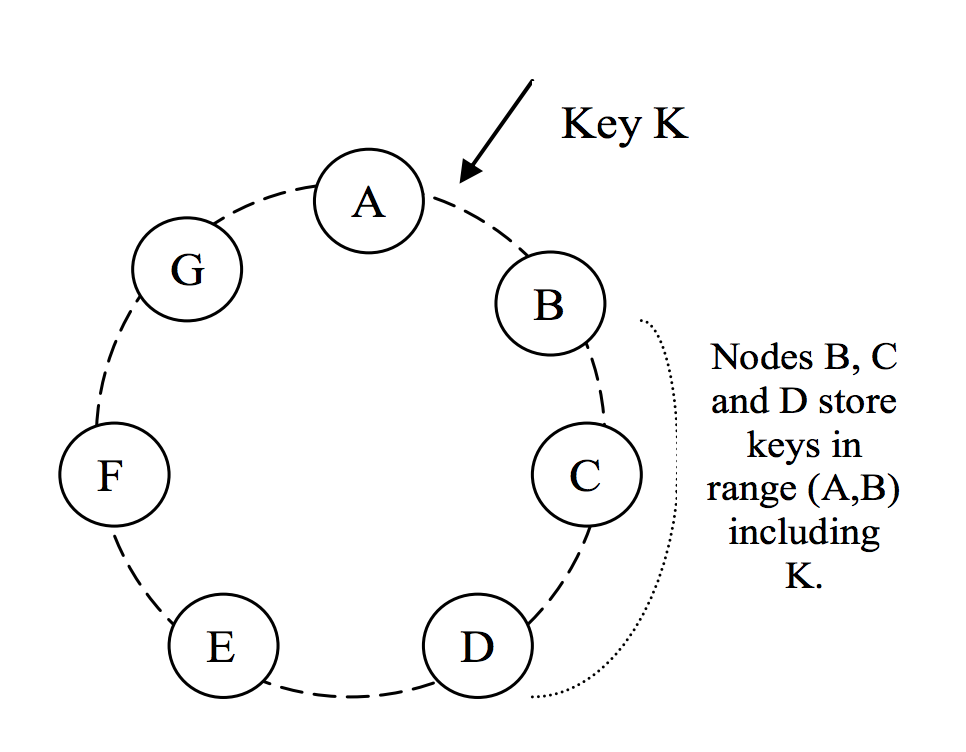
### Dynamic membership management

* Storage nodes can come and go
  + Node **“X”** join: some keys are assigned to “X”’s successor now assign to X
  + Node “X” leave: all keys are assigned to “X” now assigned to “X” successor

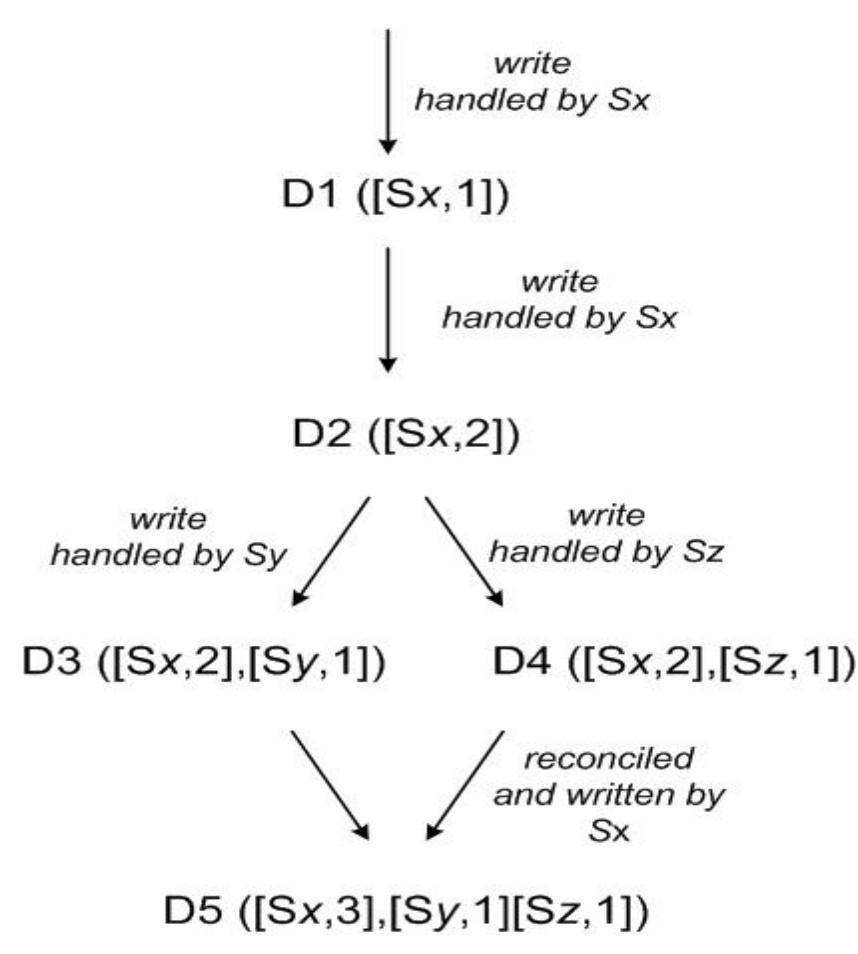
### Load balancing

* Each node **should be** responsible for at most (1+epsilon)K/N keys (K = keys, N = nodes)
* **Virtual nodes:** each physical storage mode mapped multiple time on the circle
  + Improve load-balancing, allow heterogeneous storage nodes have better performance

## Data replication

* GOAL: high availability and durability
* IDEA: replicate on (N-1) next nodes on circle. If virtual nodes links to existing replication node => skip that nodes.
  + Example from image: key k, coordinator node is B, replicated to N-1 other successor node (C, then D)
  + B,C,D are a **preference list** of key k

## Data Versioning

* GOAL: eventually consistency
* Data replication performed after ACK is sent to client **put** request
  + Asynchronous replication
  + May result in inconsistencies by partition => read not return latest value
* Operation shouldn’t be lost. If cannot accessing latest version => create different version
  + After a partition heal => merge versions
  + **Application must be designed for this merge**

### Key technique: Vector clocks

* Theory: each “write” to key k associated with vector clock VC(k)
  + VC(k) is list of (node, counter)
  + Counter “x” increase every time node\_x handle for write
  + Vetor clocks associated with every version of every nodes
* Practice: using reference list to create a list, by partition k can be written in nodes not in a list => should truncate oldest if more than threshold
* IDEA: if there are 2 version of object on parallel branch => require reconciliation by using vector clocks

## Execution “get” and “put” operation

* Storage nodes can receive request for any key
  + Genetic load balancer choose random node
  + Application can directly contact coordinator in the preference list ( every node keeps this list)
* Request routing
  + Serve request if node in preference list
  + If not, chose the first node in preference list
* Maintain consistencies of replica => quorum
  + Quorum system R+W > N (R: nb of nodes for get operation, W nb of nodes for put operation, N nb of replica node)
  + Handling put (by coordinator): generate new VC => send value (VC) to N nodes from preference list => wait W-1 Ackowledgement
  + Handling get (by coordinator): send N selected node from preference list, wait for R responses => select highest version of VC, reconcile and merge different version => write back
  + R,W should be < N => for latency, because it based on lowest of the R or W replica

## Handling failures

* All read and write operation are performed on first N healthy list (not N nodes encountered by following the ring)
  + Sloppy quorum
  + Strictly quorum in previous section is not available for server failure or network partition
* Handling Temporary Failures : hinted handoff
  + If a replica in the preference list is down, then a new replica is created on a new node
  + Coordinator selects a new replica node, but hints that the role is temporary
  + When the new replica learns about failure recovery, it handles data to the node in the preference list

## Anti-Entropy Synchronization

* GOAL: threat to durability, keep replicas synchronized
* Detect inconsistencies and minimize amount of transfer data => Merkle tree
* MERKLE TREE: every non-leaf node is labelled with hash of the labels of its children node. Leaves are hashes of the values of particular key

### Storage node:

* + Keep a merkle tree for each its key ranges (list of covered keys)
  + Compare root of the tree with replica
  + If equal => already sync. If not, traverse tree to synchronize the key that differ

## Membership management

* Gossip protocol to propagate membership changes
  + Nodes contact a random node every second
  + 2 nodes reconcile membership information
  + Gossiping also used to handle metadata
* Failure detection
  + Local notion of failure detection: with steady load on A, A periodically check status of the node in extended list
  + Does not make distinction between **failures and partitions**

# HBASE

* RDBMS: random access, structure data
* Hadoop and mapreduce: unstructured data, batch processing (large file storage, streaming access)

## Column-oriented Databases

* Data layout: save their data by columns on disk (RDBMS store by rows)
* Special databases for specific workload:
  + Reduced I/O, Real time access
  + Better for compression (column value often every similar)
* HBase is not column oriented database, use on-disk column storage format => use key-based to access cells of data, sequential range of cells

## Building block

* BigTable: distributed storage system for managing structure data, designed to scale to very large size. It is sparse, distributed, persistent multi-dimensional sorted map
* Hbase : open-source version of bigTable

### Building block: Tables – Rows – columns – Cells

* + Basic unit of HBase is column: each column has multiple version, distinct value stored in cells (can be compressed or tagged to stay in memory)
  + One or more columns form a row and addressed uniquely by row key (unique, any arbitrary array of bytes, comparable)
  + Column families:
    - Columns are grouped into coulumn families based on semantical boundaries
    - Columns families and columns stored together in low-level storage file, H-file
    - Defined when table created and cannot changed too often
    - Number of column family should be reasonably – can fit into memories
  + Null values cells or columns are not stored
  + Cells, columns value is timestamped (keep many version that changes overtime)
  + Cell versions can constrained by predicate deletion (boolean expression – Ex: keep value on the last week)

### Auto Sharding

* Region: basic unit for load balancing and scalability
  + Store “contiguous range of rows” together => range partitions in RDBMS
  + Regions will be dynamically splitted when too large, and can be merged to reduce number of files
  + In practice: System monitors region size. If the threshold is attained => SPLIT region in two at “middle key” (equivalent in size)
* Region server:
  + Each region serve 1 region server: nb of regions depend on single server (how good they can handle)
  + Server failure: regions allow for fast recovery, fine-graned load balancing when regions can move across servers

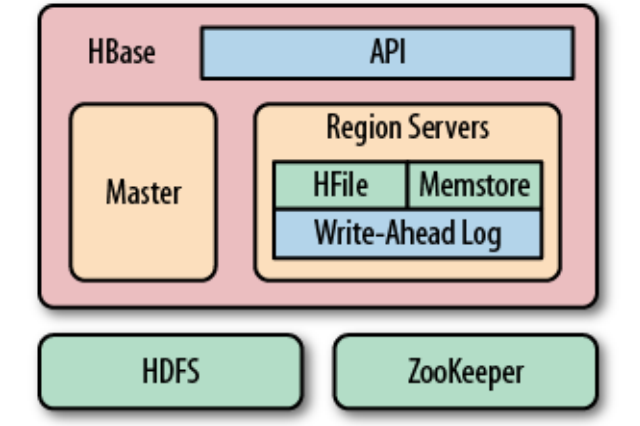
### Storage API

* Not support for Structured Query Language but support simple data model with dynamic control over data layout + format
* Scan API: fast iteration over range of rows, limit number and which column will be returned, control version number of each cell
* Read-modify-write API: single-row transactions. Implement **atomic** read-modify-write sequence under single row key
* Coprocessor: store “client supplied code” in address space of server => implement batch job, analysis data

### Hbase implement

* Data storage:
  + Hfile, persistent and order immutable map from key to value
    - Sequence of blocks with the index at the end. The index kept in memory after Hfile is loaded
* Data lookup
  + Block index => look up in single disk seek.
  + First, Binary search in the in-memory block index => check contain given key
  + Then, Read block to find atual key
* Implemented on top of HDFS
* WRITE OPERATION
  + First, data is written on commit blog (write-head-log <~ use for recovery)
  + Then data move to memory, in structure called “menstore”
  + When the size of “menstore” exceed => HDFS file
  + Rolling mechanism (Write while serving READ and WRITE): new empty slot in “menstore” take the updates while older spill to disk
  + Data in “menstore” already sorted by key
* Data locality
  + Intelligent of key design, sorted key in Hfile => we can keep data close together
* Deleting data
  + HFile is immutable => cannot delete, use “delete marker” to indicate
  + Compactions will finalize delete process
* READ operation
  + Involve in data in mem-store and data on disks
  + Many API for read and scan
* Compaction
  + Flush data from “memstore” => more small Hfile -> merge
  + Minor compaction: smaller files merge into larger files. Sorted-key file => bound by I/O performance
  + Major compaction: rewrite all files with in column family or region into one new file.
    - Drop deletion data. Perform predicate deletion (example: remove old data)

## Architecture

* Master node:
  + Assign region to region server using Zookeeper, load balancing between region server
  + Not a part of data storage or retrival path process
  + Hold meta data and schema
* Region Server:
  + Handle READ and WRITE
  + Region splitting

## SEEK and TRANSFER

### B-tree

* Efficient lookup, insert, deletion of records defined by key
  + Dynamic and multiple-index
  + B-tree vs Hash table: tree optimization, re-organization and balancing (costly)
* Support for range scans
  + Leaves are linked and represent in-order list of all keys
  + No costly tree-traversal

### LSM-tree

* Dataflow:
  + Incoming data -> store sequentially in log file
  + When log has modification saved => push to memory
  + When memory is full -> spill to disk as the sorted list (key -> value)
  + Then log file can be throw away
* Stores file
  + Arrange like B+ tree, but optimized for sequential disk access
  + All nodes of the tree try to be filled up completely
  + Update done by “rolling merge” -> packs existing on-disk multipage blocks together with in-memory
* Clean-up: background process -> aggregate files into larger file to limit disk seek
* Data look-up: memory-store first and on-disk later
* Delete data: delete marker (delete record when re-write page), predicate deletion

|  |  |
| --- | --- |
| B-tree | LSM tree |
| * Work well if no many update * More and faster you insert data at random location. The faster page get fragmented * Updates and delete are done seek rates rather than transfer rate | * Work at disk transfer rate => better for huge amount of data * Consistent insert rate (sequential write rather than random) * Reads are independent on write * Optimized layout => predictable boundaries on disk seek |

# Storage

# Cassandra

## Intro

Cassandra tries to combine the basic techniques of HBase and Dynamo.

* It’s a column-oriented **data model** (one key per row, columns and column family), here data is stored in order and the keys are assigned subsequently to each node this allows us to do some operation on data more easily;
* **Consistency** policy can be change with this feature we can have a AP or CP system.
* **Partitioning** can be chosen (Random or ByteOrdered partitioner) and can be changed on fly but this means the entire rewrite of all data so it has to be carefully chosen
  + **Random Partitioner:** works very similar to Dynamo, it associate to each data a key identifier and an identifier to each node. It uses consistent hashing and data monitoring in order to have load balancing between the nodes
  + **ByteOrderedPartitioner: supports range queries**, to do so row keys must be stored in order, the keys are splitted lexicographically among the nodes in the ring**.**

## Replication

Replication is asynchronous, the coordinator choose N-1 successor nodes clockwise in the ring. When the network topology is known than the replica can be located in different datacenters or racks (like HDFS)

#### Snitches

**Snitches**: a snitch determines which data centers and racks nodes belong to. Snitches inform Cassandra about the network topology so that requests are routed efficiently and allows Cassandra to distribute replicas by grouping machines into data centers and racks. Specifically, the replication strategy places the replicas based on the information provided by the new snitch. All nodes must return to the same rack and data center. Cassandra does its best not to have more than one replica on the same rack (which is not necessarily a physical location).

## File system

Cassandra by default stores it’s data on **HDFS** (but another FS can be configured). The problem with HDFS is that it doesn’t allows to delete or modify data, it just allows to append. So cassandra appends new info and at read time it just take the newer one. Then, an housekeeper is in charge to clean up data it reads it look if sono info was appended and in this case deletes the file and writes back just the newer one.

## Data model: special columns

* Counter columns: are used to store counters/timestamps associated to a row
* Expiring columns: specify a time to live after which data is deleted
* Super columns: is a multiple column but on just one lookup value (ex Address->(City,Zip code,...))
* SSTables: sorted string table is an immutable data file to which Cassandra writes memtables periodically. SSTables are stored on disk sequentially and maintained for each Cassandra table.

## Read and write operations

* I/O request are proxy based: routed by a coordinator which routes the request to any replica.
* The proxy node is in charge to handle the interaction between Cassandra and the client determine which are the replica nodes.

#### Write request

Is similar to HBase:

* Write commit log,
* write in memory data structure (memtable)
* write is considered SUCCESSFUL
* writes are flushed to disk in SSTable

## Flushing

* The process of turning a Memtable into a SSTable is called **flushing**. You can manually trigger flush via jmx (e.g. with bin/nodetool), which you may want to do before restarting nodes since it will reduce CommitLog replay time. Memtables are sorted by key and then written out sequentially. Thus, writes are extremely fast, costing only a commitlog append and an amortized sequential write for the flush.
* Once flushed, SSTable files are immutable; no further writes may be done. So, on the read path, the server must (potentially, although it uses tricks like bloom filters to avoid doing so unnecessarily) combine row fragments from all the SSTables on disk, as well as any unflushed Memtables, to produce the requested data.
* To bound the number of SSTable files that must be consulted on reads, and to reclaim space taken by unused data, Cassandra performs compactions: merging multiple old SSTable files into a single new one. Compaction strategies are pluggable; out of the box are provided SizeTieredCompactionStrategy, which combines sstables of similar sizes, and LeveledCompactionStrategy, which sorts sstables into a hierarchy of levels, each an order of magnitude larger than the previous. As a rule of thumb, SizeTiered is better for write-intensive workloads, and Leveled better for read-intensive.
* (For those familiar with other LSM implementations, it's worth noting that Cassandra can remove tombstones without a "major" compaction combining all sstables into a single file.)
* Since the input SSTables are all sorted by key (technically, by token), merging in a compaction can be done efficiently, again requiring no random i/o. Even so, compaction can be a fairly heavyweight operation.

## Bloom Filters

Bloom Filters are used, in SSTable, to combine row data from multiple sources and to check if a given key exists in the SSTable before reading from the disk.

A [**Bloom filter**](https://en.wikipedia.org/wiki/Bloom_filter) is a space-efficient probabilistic data structure, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not, thus a Bloom filter has a 100% recall rate. In other words, a query returns either "possibly in set" or "definitely not in set". Elements can be added to the set, but not removed. The more elements that are added to the set, the larger the probability of false positives.

## Read request

The behaviour of a read it’s similar to Dynamo. The proxy checks for inconsistency and if any resolves them and write the new data back to disk. This is done in background after the read requests has been served to the client. The number of replicas used depends on the setup it can be the closest one or all replicas and the proxy waits for a quorum.

When a node receives a read request:

* Row must be combined from all SSTables on that node
* Data not yet flushed to SSTables, i.e. stored in memtables, must be considered as well

→ This produces the requested data

In order to have high performance we use row-level column index and bloom filters.

## Consistency levels

In Cassandra consistency can be reduced in order to increase the availability and read and write consistency level can be independent.

Given N replicas in the preference list. Write request:

* all N replicas are contacted
* Ends when W respond (i.e. acknowledgment) Read request: only R replicas are contacted
* This is optimistic, may need to contact all N replicas

Choices of W and R define consistency level

Dynamo: W + R > N (recall extended preference list + sloppy quorum)

Cassandra: W + R > N not mandatory

#### Quorums

QUORUM

* W = floor(N/2 + 1): A write is written to the commit log and memtable on a quorum of W replicas
* R = floor(N/2 + 1): Read returns the record with the most recent timestamp, once a quorum of size R has responded

LOCAL\_QUORUM: Restricted to a local datacenter

EACH\_QUORUM: QUORUM invariant must be satisfied across datacenters

#### One

* W=1: one replica must write to commit log e memtable
* R=1: return the response from the closest replica and the read repair runs in background to make other replica consistent

#### ALL, ANY

* W=N: all replica must acknowledge
* R=N: returns the record with the most recent timestamp across all replica

ANY: additional consistency for writes uses hinted handoff in order to complete a write also is all replica are down