BAB 2 LANDASAN KEPUSTAKAAN

Pada bab ini berisi uraian dan pembahasan tentang teori, konsep, model, metode, atau sistem dari literatur ilmiah, yang berkaitan dengan tema, masalah, atau pertanyaan penelitian. Penjabaran pada bab ini merupakan ringkasan dari teori-teori yang menunjang permasalahan pada penelitian tersebut terhadap informasi dari sumber pustaka. Teori-teori ini yang selanjutnya akan digunakan sebagai dasar pemahaman materi berkaitan dengan permasalahan yang diangkat serta digunakan dalam penelitian. Tinjauan pustaka bersumber dari jurnal, penelitian terdahulu, dan buku.

2.1 Kajian Pustaka

Beberapa penelitian yang terkait dengan analisis kinerja pada pada algoritme penjadwalan Hadoop yang telah dilakukan sebelumnya yang dapat digunakan sebagai bahan referensi penulis dalam melakukan penelitian ini. Pada penelitian ini menggunakan pengembangan penelitian terdahulu sebagai bahan perbandingan untuk mengetahui perbedaan terhadap penelitian ini. Berikut akan diuraikan beberapa penelitian terdahulu yang berhubungan dengan algoritme penjadwalan pada Hadoop.

1. Alfian Dzulfikar K. (2015) melakukan penelitian tentang analisis algoritme FIFO dan *Capacity Scheduling* pada Hadoop. Tujuan penelitian ini adalah untuk melakukan analisis terhadap kinerja dari algoritme FIFO dan *Capacity Scheduling* dengan menggunakan parameter pembanding *Job Fail Rate, Throughput, dan response time* sebagai acuan perhitungan kinerja sistem. Pada penelitian ini menggunakan 3 jenis *job* yaitu, *job wordcount, job grep,* dan *job randomtextwritter.* Hasil penelitian menunjukkan bahwa *Capacity Scheduling* memiliki hasil kinerja yang lebih baik daripada FIFO. Nilai dari *Job Fail Rate* pada *Capacity Scheduling* menunjukkan nilai maksimal 4,3%, sedangkan pada FIFO menunjukkan 10%. Pada jenis *job grep* mengalami penurunan *Job Fail Rate* menjadi 1,1% pada total 25 *job* dan mengalami peningkatan *Job Throughput* sebesar 4 *job* pada 22,5 menit.
2. Komaratih Dian P. (2015) melakukan penelitian tentang analisis algoritme FIFO dan *Delay Scheduling* pada Hadoop. Pada penelitian ini menggunakan parameter *Job Fail Rate, Throughput, dan Response Time* sebagai bahan pembanding untuk acuan perhitungan kinerja sistem. Selain itu analisis ini menggunakan jumlah dan jenis *job* yang berbeda-beda setiap skenario sistem yang dijalankan pada masing-masing *job scheduling.* Jenis *job* yang digunakan adalah *job wordcount, job grep,* dan *job randomtextwritter.* Hasil penelitian ini menunjukkan bahwa *delay scheduling* memiliki hasil kinerja lebih baik dibanding dengan FIFO. Pada jenis *job wordcount* mengalami penurunan 0.3% pada *Job Throughput,* pada jumlah *job* 50 *job* pada *Response Time* lebih cepat 142 menit 45 detik (Komaratih,2015).
3. Tri Retno P. (2016) melakukan penelitian tentang analisis penggabungan *Fair Share Scheduling*, dan *Delay Improve Fair Share Scheduling* pada Hadoop. Pada penelitian ini menggunakan jenis *job* yaitu, *job wordcount, job grep,* dan *job*

randomtextwritter dan menggunakan parameter *Job Fail Rate, Throughput, dan response time* sebagai bahan pembanding untuk acuan perhitungan kinerja sistem. Hasil dari penelitian ini menunjukkan bahwa *Delay Improve Fair Share Scheduling* memiliki kinerja yang lebih efektif jika dibandingkan dengan *Delay Scheduling* dan *Fair Share Scheduling*. Hasil analisis ini dibuktikan dengan pada jenis *job randomtextwritter* dengan penurunan 0,3% pada *Job Fail Rate*, dengan nilai throughput 2,73 *job* /menit dan lebih cepat 273,59 menit dari *Delay Scheduling* dan lebih cepat 128,15 menit dari *Fair Share Scheduling* (Tri Retno,2016).

Perbandingan penelitian terdahulu dengan penelitian sekarang dapat dilihat pada Tabel 2.1 berikut ini.

Tabel 2.1 Perbandingan Penelitian Sebelumya

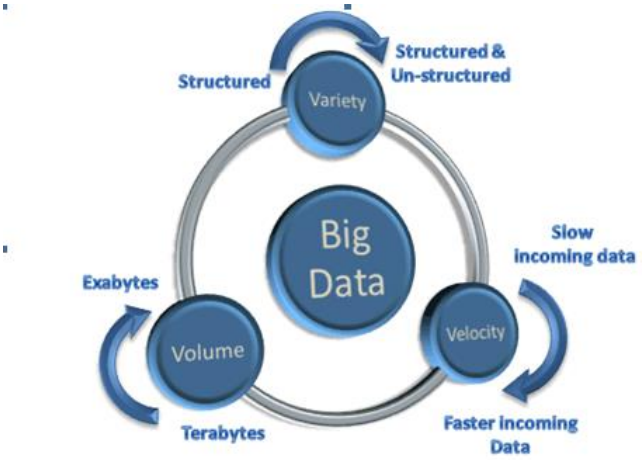
Peneliti	Algoritme *Scheduling*					Jenis *Job*			
	FIFO	*Delay*	*Fair Share*	*Delay Improve Fair Share*	*Capacity*	*Word count*	*grep*	*Random text writer*	*Word Mean*
Alfian Dzulfikar (2015)	√				√	√	√	√	
Komaratih Dian P. (2015)	√	√				√	√	√	
Tri Retno Pamungkas (2016)		√		√		√	√	√	
Penelitian ini			√		√	√			√

2.2 Konsep Big Data

Big Data merupakan istilah dari sekumpulan data yang kompleks dan berskala besar sehingga sangat sulit untuk diproses dengan cara pemrosesan data menggunakan alat konvesional. Pada awalnya McKinsey & Company adalah perusahaan pertama yang mengemukakan konsep *big data.* Pada bulan Juni 2011 McKinsey & Company mengeluarkan laporan tentang *big data* untuk dilakukan analisis secara rinci terhadap dampak dari perkembangan data yang menjadi kekhawatiran bagi industri teknologi (Zan Mo,2015). Ada dua faktor perkembangan *big data.* Faktor yang pertama adalah dari segi perkembangan

internet dan meningkatnya jumlah objek yang terhubung dan saling berkontribusi untuk membuat *volume* data berkembang pesat. Faktor kedua adalah pengembangan jumlah kapasitas penyimpanan dan komputasi data yang semakin sulit (Doug , 2001). Dalam pemrosesan *big data* terdapat 3 model atau yang lebih dikenal dengan 3V yaitu, *volume, variety* dan *velocity* (Azzeddine, 2015):

1. *Volume* : *Big data* ini mengacu pada jumlah massa data. Ukuran besarnya data ini biasanya dalam ukuran terabytes keatas. Banyak perusahaan saat ini memiliki data yang terus bertambah dari semua jenis sektor. *Volume* data akan terus meningkat sehingga diprediksi ukuran massa data akan lebih besar dari *petabyte*s hingga *zetabytes*.
2. *Variety* : Variasi dari beberapa jenis data yang akan dikelola kompleksitasnya. Jenis data pada *variety* ini meliputi *structured data*, *semi-structured*, dan *unstructured data.* Contoh jenis data pada *structured data* adalah data dengan format .xls dan .csv, pada *semi-structured* adalah pada *source code* html dan xml, pada *unstructured data* adalah .txt dan konten *audio/video*.
3. *Velocity* : Kecepatan data yang masuk bisa diukur dalam waktu per-detik, per-menit, per-jam. Kecepatan dimana data yang dibuat, diolah , dan dianalisis terus-menerus secara *real time.*



Gambar 2.1 Model 3 V pada Big Data
Sumber : Iexpertify (2014)

2.3 Hadoop

Hadoop merupakan suatu *software framework open sourc*e berbasis java dibawah lisensi Apache yang digunakan sebagai pengolah data yang berukuran besar dan desain sistem *file* secara terdistribusi. Hadoop ini berjalan diatas *cluster* yang saling terhubung dengan beberapa komputer lainnya. Hadoop disebut juga sebagai *analytic engine* yang digunakan seseorang untuk melakukan pembuatan penulisan perintah (*write)* dan menjalankan *(run)* aplikasi pemrosesan data dalam jumlah besar.

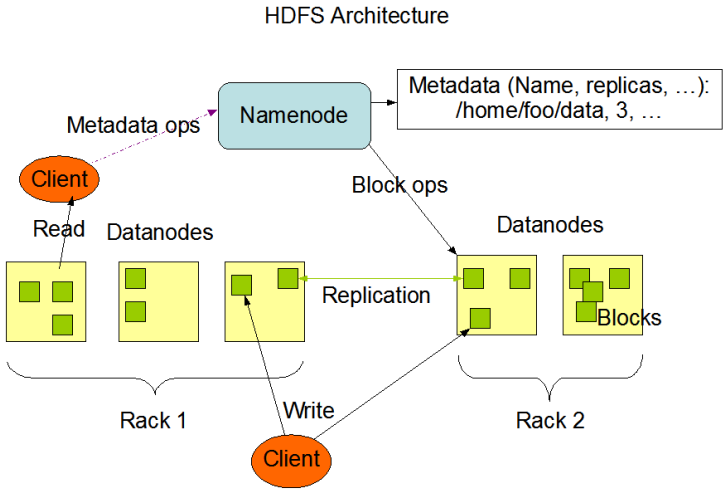
Berikut adalah komponen-komponen penyusun dalam Hadoop sebagai berikut.

1. Hadoop *Distributed File System* (HDFS): sistem *file* terdistribusi yang menyimpan data pada mesin komoditas Hadoop, serta memberikan *bandwidth* yang sangat tinggi di seluruh *cluster*.

2. Hadoop MapReduce: model pemrograman untuk pemrosesan data berskala besar.
3. Hadoop YARN: platform manajemen sumber daya yang bertanggung jawab untuk mengelola sumber daya komputasi dalam kelompok dan menggunakannya untuk aplikasi penjadwalan.

2.3.1 HDFS (Hadoop *Distributed File System)*

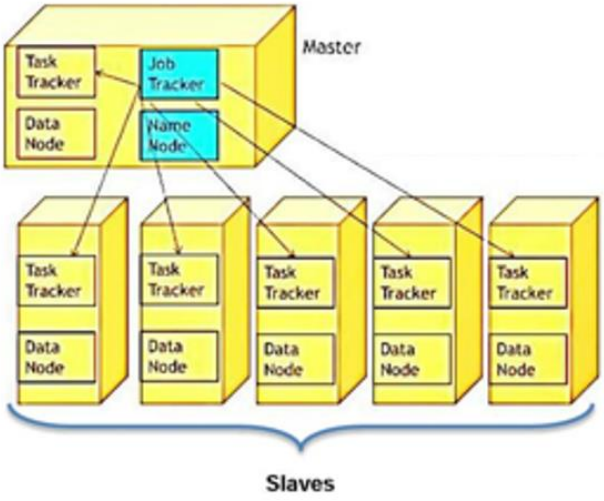
HDFS sebagai direktori komputer dimana data Hadoop disimpan. Data yang besar tersebut akan dibagi dalam beberapa *chunk* yang dikelola oleh setiap *node* pada *cluster*. Setiap *chunk* akan direplikasi sehingga jika terdapat *node* yang mengalami kegagalan maka data tersebut masih bisa diakses oleh *node* lain yang terdapat dalam *cluster* tersebut. HDFS dikembangkan menggunakan desain sistem *file* secara terdistribusi. Tidak seperti sistem terdistribusi, HDFS sangat *fault tolerant* dan dirancang menggunakan *hardware low-cost.* HDFS umumnya diklasifikasikan menjadi dua bagian yang disebut dengan name*node* dan data*node*. Name*node* terletak pada server induk yang digunakan untuk mengelola sistem *file* namespace seperti membuka, menutup, mengganti nama *file* atau direktori dan juga digunakan untuk mengatur akses *file* yang dilakukan oleh *slave* (Dhruba,2013). Name*node*s juga digunakan untuk mengelola metadata pada tempat data di *cluster* dan juga mereplikasi data tersebut. Sedangkan data*node*s terletak pada komputer *slave*. Data*node* digunakan untuk menyimpan data atau *file*. Pada data*node* dalam satu *file* dibagi menjadi beberapa blok *file* biasanya setiap blok *file* berukuran 128 MB. Setiap blok *file* akan direplikasi di data*node*s, *replica* ini diwakili oleh dua *file* dalam *file* sistem pada localhost. *File* pertama berisi data itu sendiri dan *file* kedua merupakan blok metadata yang terdapat *checksum* untuk blok data dan blok *generation stamp.* Data*node*s bertanggung jawab untuk melakukan proses *read and write* dari *client file* sistem dan melakukan pemblokiran, penghapusan, dan replikasi atas intruksi dari name*node* tersebut (Dhruba,2013).



Gambar 2.2 Arsitektur HDFS
Sumber :Dhruba (2013)

2.3.2 MapReduce

MapReduce merupakan model pemrograman yang dikembangkan oleh google. Pada awalnya MapReduce bertujuan sebagai aplikasi pencarian atau pengindeksan i*nternal* Google, tapi sekarang MapReduce digunakan secara luas oleh banyak perusahaan-perusahaan besar seperti Yahoo, Amazon, dan IBM. MapReduce sebagai sistem yang mendasari untuk melakukan partisi input data, membuat jadwal eksekusi program di beberpa mesin, *handling* kegagalan mesin, dan mengatur komunikasi yang diperlukan antar mesin. Dalam hal ini, semua kegiatan tersebut dikelola oleh *Job Tracker* dan dieksekusi oleh *TaskTracker* yang merupakan komponen utama dari MapReduce dalam memanfaatkan arsitektur *master* atau *slave*. Peran dari *Job Tracker* adalah *master* dari *TaskTracker* yang mengatur jadwal serta mengkoordinasi semua *job* yang telah di *submit* oleh *slave* dan juga menangani distribusi *job* ke *TaskTracker.* Peran dari *TaskTracker* untuk menerima *job* dari *Job Tracker* dan memutuskan *job* tersebut kedalam *mapreduce task. TaskTracker* juga melakukan eksekusi tugas dan melaporkan status terbaru kepada *Job Tracker* dengan mengirimkan pesan *heartbeat* dan mengirimkan hasil terakhirnya (Ruchi, 2015). Berikut merupakan urutan cara kerja MapReduce pada komputer *master* dan komputer *slave* pada gambar 2.3.



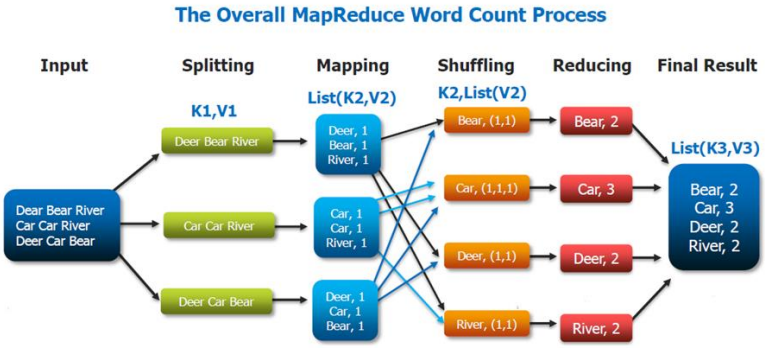
Gambar 2.3 Cara kerja MapReduce pada Pengiriman *Job*

1. Pada komputer *slave*, pengguna melakukan *submit job* ke *JobTracker.*
2. Kemudian *Job Tracker* bertanya kepada *Namenode* untuk mengetahui letak lokasi dari data.
3. Setelah mengetahui letak lokasi data, kemudian *JobTracker* meminta *TaskTracker* menjalankan tugas terhadap data yang ada pada masing-masing *node*.
4. Setelah *TaskTracker* berhasil menjalankan tugas, hasil pengolahan data disimpan dalam *Datanode* dan *Namenode* diberitahu letak lokasi data pada *node* mana yang berhasil disimpan.
5. *TaskTracker* memberitahu *JobTracker* bahwa tugas telah berhasil dijalankan.

6. *JobTracker* menginformasikan pada pengguna bahwa tugas telah berhasil dijalankan.

Selain itu pada MapReduce terdapat tiga cara kerja pemrograman yang dibagi sebagai berikut (Samira, 2012) :

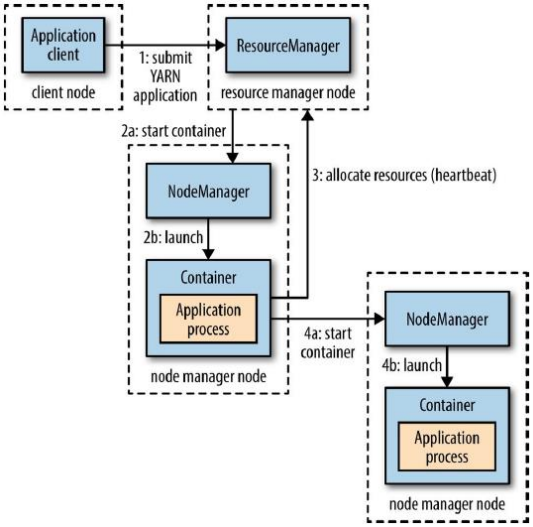
1. *Map* : Pada tahap map ini merupakan tahap pertama yang digunakan untuk meng-*extract* data kedalam fungsi *Map* (*Mapper*), setiap *Mapper* berjalan secara paralel. Keluaran berupa kumpulan pasangan dari <*key,value*>.
2. *Sort and Shuffle* : Keluaran dari setiap *Mapper* dipartisi oleh kunci *hashing*. Pada fase ini jumlah partisi sama dengan jumlah reduksi. Setelah dipartisi dalam fase *map*, setiap partisi disimpan oleh *key* yang digunakan untuk menggabungkan *value* berdasarkan key yang sama.
3. *Reduce* : Semua *value* yang mempunyai *key* yang sama akan di *reduce,* diaggregasi, dan di *summary*. Keluarannya berupa *<key, value>* yang telah dilakukan proses pada *reducer* yang telah dikelompokkan.



Gambar 2.4 Cara Kerja Pemrograman pada MapReduce
Sumber :Tatiana (2017)

2.3.3 YARN *(Yet Another Resource Negotiator)*

YARN merupakan salah satu komponen tambahan yang terdapat pada Hadoop versi 2. YARN merupakan sistem manajemen sumber daya yang secara umum disediakan oleh Hadoop untuk meningkatkan kemampuan MapReduce. Komponen utama yang terdapat pada YARN yaitu, ResourceManager dan *Node*Manager. Pada ResourceManager mengatur penjadwalan *cluster* untuk mengalokasikan *resource* ke berbagai *job* yang sedang berjalan sesuai dengan batasan kapasitas antrian, batas pengguna. Pada *Node*Manager bertugas untuk menjalankan memantau *container,* memantau penggunaan *resource* (CPU, memori, *disk*, jaringan) dan melaporkan ke *ResourceManager* oleh semua *node* yang terdapat pada *cluster.* Setiap *container* menjalankan aplikasi yang lebih spesifik dengan batasan pada pemakaian memori, CPU, dan sebagainya (Tom White, 2015). Pada gambar 2.4 menggambarkan bagaimana YARN berjalan pada sebuah aplikasi.

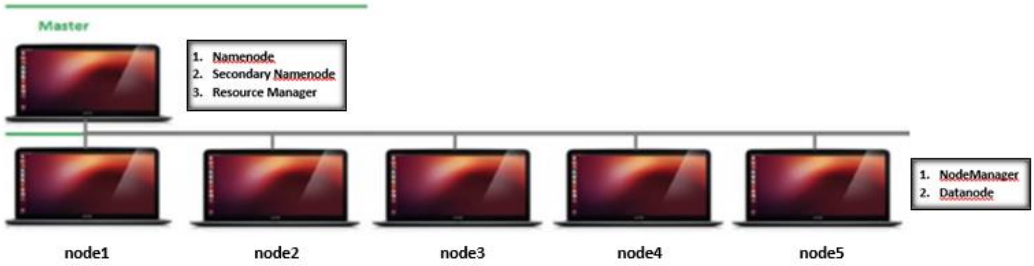


Gambar 2.5 Cara Kerja YARN pada Sebuah Aplikasi
Sumber :Tom White (2015)

Pada langkah 1 untuk memulai menjalankan aplikasi pada YARN, *slave* akan melakukan komunikasi dengan *resource manager* untuk menjalankan proses aplikasi pada *master*. Kemudian pada langkah 2a dan 2b, *resource manager* menemukan *node manager* yang akan melakukan *launch* untuk melakukan proses aplikasi *master* kedalam *container.*Pada langkah ketiga pada *application master* tidak hanya sekali saja berjalan pada proses, tergantng pada aplikasinya. Pada *application process* melakukan perhitungan pada *container* yang sedang berjalan dan akan mengembalikan hasilnya kepada *client.* Selain itu pada proses 4a dan 4b hasil dari application proses dapat digunakan untuk menjalankan perhitungan terdistribusi.

2.4 Hadoop *Multinode*

Hadoop *multi node* menggunakan gabungan dari 2 mesin *server single node*, 1 mesin untuk komputer *master* dan 1 mesin untuk komputer *slave*. Pada Hadoop *multi node* ini, komputer *slave* menggunakan 5 buah komputer. Sehingga pada Hadoop multi*node* ini dapat melakukan pengiriman *job* oleh beberapa pengguna secara bersama-sama. Pada layer HDFS komputer *master* ini menjalankan *namenode, jps, SecondaryNameNode,* dan *ResourceManager.* Sedangkan pada komputer *slave* menjalankan *datanode, jps,* dan *NodeManager.* Kedua mesin ini komputer *master* dan komputer *slave* dikonfigurasi melalui beberapa *file* yaitu, hdfs-site.xml, core-site.xml, mapred-site.xml, dan yarn-site.xml. Pada gambar 2.5 merupakan gambaran topologi Hadoop multi*node* user sebagai berikut.



Gambar 2.6 Topologi Hadoop *Multinode*

2.5 Algoritme S*cheduling*

Pada Hadoop, *job* yang telah di-*submit* oleh pengguna akan saling berkompetisi untuk merebutkan suatu *resource* yang tersedia, dalam metode ini akan dicatat penggunaan waktu *processor* pada saat melakukan beberapa proses. Untuk itu perlu adanya algoritme penjadwalan yang bertugas mengatur jalannya *job* pada sebuah *resource* serta mengelola pemrosesan data agar *output* yang dikeluarkan sesuai dengan apa yang diharapkan. Dengan menggunakan algoritme penjadwalan memiliki kelebihan tersendiri diantaranya dapat menggunakan waktu *processor* secara efisien, dapat meminimalkan *overhead*, dan dapat memaksimalkan *Throughput*.

Job yang masuk dalam sebuah antrian dapat diatur menggunakan beberapa algoritme yang ada pada Hadoop diantaranya algoritme *First In First Out (FIFO) scheduling, Capacity Scheduling, Fair Share Scheduling, Deadline Constraint scheduling, Resource Aware scheduling,* dll. Pada masing-masing algoritme tersebut terdapat karakteristik yang berbeda-beda serta memiliki kelemahan dan kelebihan tersendiri yang dapat menjadi bahan pertimbangan dalam penggunaannya.

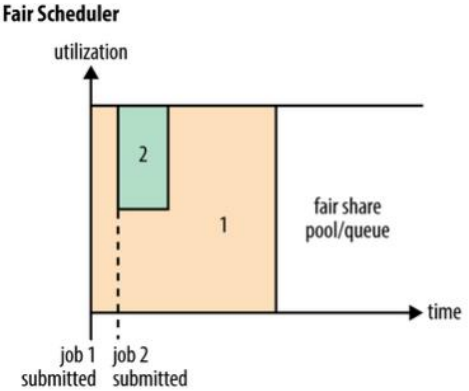
2.5.1 Algoritme *Fair Share Scheduling*

Fair Share Scheduling merupakan metode yang digunakan untuk setiap *job* yang berjalan dalam *cluster* Hadoop agar mendapatkan *resource* yang sama dengan *job* yang lainnya. Dengan metode ini maka *short job* akan lebih cepat dieksekusi dan tidak membutuhkan waktu yang lama pada resource yang sedang digunakan untuk *long job.* Setiap pengguna akan memiliki nilai minimum *share* yang dapat digunakan untuk mengsubmit *job*. Nilai minimum ini akan masuk ke slot *resource* yang mempunyai batas nilai. Setiap *job* yang selesai dikerjakan akan keluar dari slot, sehingga *resource* akan memiliki slot yang kosong dan bisa digunakan pengguna untuk mengsubmit *job* kembali. Dalam *Fair Share Scheduling* memungkinkan *job* untuk dijalankan secara default, tetapi juga memungkinkan untuk melakukan pembatasan pengguna yang sedang menjalankan *job* melalui *file* yang telah dikonfigurasi. Pembatasan *job* tersebut berguna jika pengguna yang mengirimkan *job* kedalam *cluster* dengan jumlah yang sangat banyak. Karena

terlalu banyak data sehingga mengakibatkan kinerja dari *Fair Share Scheduling* semakin melambat. Dengan melakukan pembatasan *job* ini tidak menyebabkan *job* tersebut gagal, hanya saja harus menunggu dalam antrian sampai beberapa *job* dari pengguna sebelumnya telah selesai dilakukan (Apache,2016). *Fair Share Scheduling* memiliki 3 konsep dasar yaitu.

1. *Job* akan ditepatkan kedalam *pool* berdasarkan dengan atribut yang telah dikonfigurasikan seperti nama pengguna, kelompok UNIX, atau penanda khusus melalui *job* conf.
2. Setiap *pool* dapat memiliki *slot* kapasitas yang ditentukan melalui *file* yang telah dikonfigurasi dengan memberikan jumlah *slot* yang minimum pada *map* dan *reduce slot* untuk dapat dialokasikan ke *pool*. Ketika ada suatu *job* yang tertunda di *poo*l, maka secara default masih terdapat *slo*t yang telah disediakan. Akan tetapi jika dalam keadaan *idle* atau sedang tidak menjalankan sebuah *job slo*t tersebut akan digunakan oleh *pool* lain.
3. Dengan kelebihan kapasitas maka *pool* tersebut dialokasikan ke *job* dengan menggunakan suatu algoritme *Fair Share Scheduling.* Pada algoritme ini memastikan bahwa setiap *job* akan menerima jumlah *resource* yang sama dari waktu ke waktu. S*hort job* akan lebih cepat selesai karena lebih diprioritaskan (Divya,2015).

Untuk lebih memahami kinerja dari algoritme *Fair Share Scheduling*, pada gambar 2.7 akan menjelaskan bagaimana *job* mendapat *resource* yang sama dalam satu *cluster*. Pada gambar tersebut terdapat satu jenis *pool* atau *queue*. *Fair Share Scheduling* adalah metode alokasi pembagian *resource* pada antrian secara adil untuk seluruh *job* yang masuk pada antrian. Ketika pengguna men-*submit job* 1, maka *job* tersebut akan menggunakan semua *resource* yang tersedia karena tidak terdapat permintaan pada *job* lain. Kemudian pengguna mensubmit *job* 2 pada antrian meskipun *job* 1 masih berjalan, maka setelah beberapa saat pada masing-masing *job* akan menggunakan setengah dari *resource* yang tersedia. Jadi setiap *job* pada *queue* akan mendapat setengah dari total keseluruhan *resource* yang ada pada *cluster* tersebut.



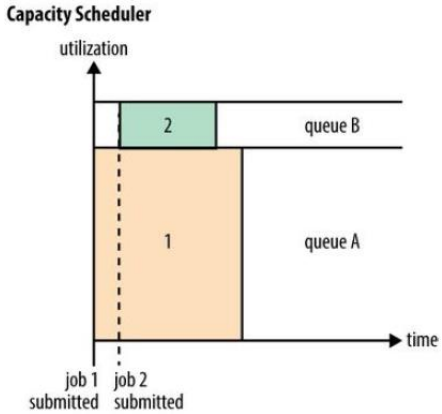
Gambar 2.7 Algoritme *Job Fair Share Scheduling*
Sumber : Tom White (2015)

2.5.2 Algoritme *Capacity Scheduling*

Capacity Scheduling merupakan sebuah algoritme penjadwalan pada YARN yang terdapat fitur *queuing priority, sub-queue,* dan dan prediksi kesalahan pada *job*. Pada suatu *queue* memiliki prioritas untuk mengakses resource dalam *cluster*. Ketika slot dalam *queue* dalam keadaan idle, maka *job* langsung dapat menggunakan *queue* tersebut meskipun *queue* tersebut tidak memiliki prioritas *job*. Kemudian ketika pengguna melakukan submit *job* dengan *queue* priority, maka *queue* yang memiliki prioritas tersebut akan langsung mendapatkan slot dari slot *queue* yang telah dipakai oleh *job* yang tidak memiliki prioritas dengan menghentikan layanan terhadap *job* tersebut (Jagmohan, 2012).

Capacity Scheduling mendukung antrian secara hirarki untuk memastikan bahwa *resource* tersebut dibagi di antara sub-antrian sebelum antrian lain yang diperbolehkan untuk menggunakan *resource* yang kosong. Gagasan utama dalam algoritme ini bahwa *resource* yang tersedia pada Hadoop *cluster* akan dibagi beberapa *cluster* secara kolektif berdasarkan pada kebutuhan komputasi. Manfaat lain dari algoritme ini dapat mengakses kapasitas dari *resource* yang sedang tidak digunakan. Manfaat ini untuk memberikan elastisitas dalam *cost-effective. Capacity Scheduling* menyediakan batas *slot* untuk memastikan bahwa suatu antrian tidak dapat memakan jumlah *resource* yang tidak sesuai dalam sebuah *cluster*. Selain itu, *Capacity Scheduling* dapat memberikan batas antrian yang diinisialisasi pada pengguna untuk menjamin stabilitas *cluster* (Divya,2015).

Pada gambar 2.8 menunjukkan proses pengiriman *job* yang masuk dalam suatu antrian pada pada algoritme *Capacity Scheduling*.



Gambar 2.8 algoritme *Job Capacity Scheduling*
Sumber : Tom White (2015)

Berikut adalah fitur-fitur yang terdapat dalam *fair-share scheduling* yaitu.

1. *Hierarchical Queue*s : digunakan untuk membagi dan menyediakan *resource* untuk sub-antrian pada sebuah organisasi sebelum antrian lain menggunakan *resource* tersebut.
2. Capacity Guarantees : Semua *job* yang di-*submit* ke antrian akan memiliki akses kapasitas yang dialokasian ke antrian.

3. *Security* : Setiap antrian memiliki *Access Control List* (ACL) untuk mengontrol pengguna yang dapat menggunakan antrian tersebut. Penguna dipastikan tidak bisa melihat dan memodifikasi aplikasi dari pengguna lain.
4. *Elasticity* : *resource* yang diberikan secara bebas dan dialokasikan untuk setiap antrian bisa melebihi kapasita yang telah ditentukan. Dalam hal ini ketika ada permintaan *resource* dari antrian yang sedang berjalan yang tidak memenuhi pada *future point in time*. Ketika antrian tersebut selesai atau dalam keadaan idle, maka *resource* akan ditugaskan untuk antrian lain.
5. *Multi-tenancy* : batas setting komprehensif untuk mencegah satu aplikasi, pengguna, dan antrian memonopoli *resource* dari antrian lain atau *cluster* secara keseluruhan.
6. *Operadibility* terdiri dari beberapa komponen yaitu.
    a. Run*time* configuration : konsol yang disediakan untuk pengguna dan administrator melihat alokasi *resource* dalam antrian pada *cluster*. Administrator dapat menambah antrian pada saat *runtime,* tetapi antrian tidak dapat dihapus pada saat *runtime.*
    b. *Drain Applications* : Administrator dapat menghentikan antrian pada saat *runtime* untuk memastikan bahwa aplikasi yang dijalankan dapat berjalan sampai selesai dan tidak aplikasi baru yang diajukan kedalam antrian tersebut.
    7. *Resource*-based *scheduling* : aplikasi dapat menentukan *resource* lebih dari default. Saat ini memori adalah kebutuhan yang mendukung *resource*.
    8. *Queue mapping based on user or* group : fitur ini digunakan untuk pengguna dapat menempatkan aplikasi pada antrian tertentu berdasarkan pada pengguna atau kelompok.

2.6 Parameter pengujian

Parameter pengujian yang dipakai untuk membandingkan kinerja dari algoritme *Fair Share Scheduling* dan *Capacity Scheduling* ini adalah parameter *Job Fail Rate, Latency,* dan *Throughput* yang akan dijelaskan sebagai berikut :

1.6.1 Job Fail Rate

Pada parameter *Job Fail Rate* ini digunakan untuk melakukan pengukuran jumlah *job* yang mengalami kegagalan. *Job* yang gagal tersebut akan terus diulang kembali prosesnya. Berikut adalah rumus yang digunakan untuk menghitung nil*ai fail rate* yaitu.

$$\textit{Fail rate} = \frac{\lambda_1 \, x \, timeout + \, \lambda_2 \, x \, error}{sum}$$

Dimana λ_1 merupakan bobot dari *timeout* dengan nilai 0,5. *Timeout* merupakan jumlah waktu dimana *job* mengalami kegagalan. λ_2 merupakan bobot dari *error* dengan nilai 1. *Error* merupakan jumlah *job* yang gagal dan diulang

kembali. Sedangkan *sum* merupakan jumlah keseluruhan dari *job* yang telah dimasukan dalam *server* Hadoop (Yang XIA, 2011).

1.6.2 Latency

Pada parameter ini digunakan untuk mengukur rata-rata waktu yang dibutuhkan suatu *job* untuk dikerjakan dari client ke server dalam suatu antrian. Nilai yang dihasilkan untuk *Latency* ini diperoleh dari waktu awal suatu *job* dikerjakan sampai waktu *job* selesai dikerjakan dan dibagi oleh jumlah *job* yang telah ditentukan. Satuan yang digunakan dalam *Latency* ini adalah menit.

$$Latency = \frac{Jumlah\ total\ waktu\ keseluruhan\ job\ (menit)}{jumlah\ job}$$

1.6.3 Throughput

Pada parameter ini digunakan untuk mengukur kecepatan rata-rata data yang diterima oleh suatu *node* dalam selang waktu tertentu. Parameter ini akan diperoleh dari ukuran data yang dikirim dibagi dengan waktu pengiriman data secara keseluruhan. Satuan yang digunakan untuk menghitung nilai *Througput* adalah bit/sec.

$$Throughput = \frac{\text{Ukuran data yang diterima (Bytes)}}{\text{Waktu pengiriman data (sec)}}$$