Reduksi Big Data dengan Algoritma Clustering untuk Spark

Matthew Ariel Wangsit–2015730010

1 Data Skripsi

Pembimbing utama/tunggal: **Veronica Sri Moertini**
Pembimbing pendamping: -
Kode Topik : **VSM4502**
Topik ini sudah dikerjakan selama : **1** semester
Pengambilan pertama kali topik ini pada : Semester **45 - Ganjil 18/19**
Pengambilan pertama kali topik ini di kuliah : **Skripsi 1**
Tipe Laporan : **B -** Dokumen untuk reviewer pada presentasi dan **review Skripsi 1**

2 Latar Belakang

Big data adalah sebuah istilah yang menggabarkan volume data yang besar, baik data yang terstruktur maupun data yang tidak terstruktur. Data-data tersebut memiliki potensi untuk digalih menjadi informasi yang penting. Dalam bidang *big data* ada berapa tantangan seperti volume data yang besar, kecepatan aliran data masuk yang harus ditangani, dan variasi data dengan format yang berbeda. Tantangan tersebut membuat *traditional data-processing application software* tidak bisa memproses dan menganalisis *big data*. Munculah teknologi-teknologi seperti Hadoop dan Spark yang khusus untuk menangani *big data*.

Big data akan lebih mudah dianalisis dan diterapkan teknik-teknik *data-mining* ketika volume *big data* tersebut telah direduksi. Dengan mereduksi data, kita bisa menghemat biaya transfer data, *disk space*, dan jumlah data yang diproses. Hasil dari reduksi *big data* harus bisa mewakili *the original big data* secara akurat.

Salah satu cara mereduksi data adalah dengan menggunakan algoritma *Clustering Agglomerative*. Algoritma tersebut cocok untuk data yang tidak memiliki atribut yang terlalu banyak. *Science journal* berjudul *Big Data Reduction Technique using Parallel Hierarchical Agglomerative Clustering* sudah meneliti algorima *Clustering Agglomerative* berbasis MapReduce pada Hadoop. Sudah terbukti bahwa data yang direduksi dengan algoritma tersebut bisa mewakili data secara keseluruhan. Algoritma *Clustering Agglomerative* bekerja dengan membiarkan setiap objek membentuk *sub-cluster*. Kemudian, *sub-cluster* akan di gabung dengan *sub-cluster* lainya secara iterasi sampai terbentuknya *single cluster*. *single cluster* akan menjadi akar dari *hierarchy*, *cluster* tersebut digabung dengan *cluster* terdekat bedasarkan jarak minimum atau kesamaan.

Walau reduksi data dengan algoritma *Agglomerative* berbasis MapReduce pada Hadoop bisa mewakili *the original big data* secara akurat, MapReduce pada Hadoop memiliki kekurangan. Kekurangan MapReduce adalah dalam melakukan *itterative processing*, MapReduce akan menuliskan hasil sementara iterasi kepada disk. Hal ini membuat MapReduce lama dalam mengerjakan proses iterasi. Algoritma *Agglomerative* yang mengandung banyak iterasi kurang baik ketika diimplementasikan pada Hadoop MapReduce, hasil sementara akan dituliskan kepada disk berulang kali.

Spark adalah *distributed cluster-computing framework* yang bisa menggantikan MapReduce beserta kekurangannya. *In-memory processing* Spark mengalahkan kecepatan pemrosesan pada Hadoop MapReduce. Karena proses dilakukan pada RAM, kecepatan pemrosesan akan jauh lebih cepat. Tidak hanya itu, kecepatan proses iterasi meningkat karena hasil sementara tidak harus ditulis kepada disk. Spark *Resilient Distributed Datasets* (RDDs) memungkinkan *multi map operation* pada memori.

Pada skripsi ini, akan dibangun sebuah perangkat lunak yang dapat mereduksi *big data*. Perangkat lunak tersebut akan dibangun menggunakan framework terdistribusi Spark dan mengimplementasikan algoritma *Agglomerative* yang khusus dikustomisasi untuk lingkungan Spark. Perangkat lunak akan menampilkan hasil reduksi dalam format visual dan tabel. Dengan menggunakan Spark, waktu proses reduksi data akan lebih cepat dibanding MapReduce.

3 Rumusan Masalah

Dari latar belakang di atas maka dapat dibentuk rumusan masalah adalah sebagai berikut:

1. Bagaimana cara kerja algoritma *Agglomerative Clustering* berbasis MapReduce untuk mereduksi *big data*?
2. Bagaimana cara mengkustomisasi dan mengimplementasikan aggloritma *Agglomerative Clustering* pada sistem tersebar Spark?
3. Bagaimana mengukur kinerja hasil dari implementasi dari algoritma *Agglomerative Clustering* pada sistem tersebar Spark?
4. Bagaimana cara mempresentasikan data yang telah direduksi agar dapat diinterpretasikan pengguna dengan mudah?

4 Tujuan

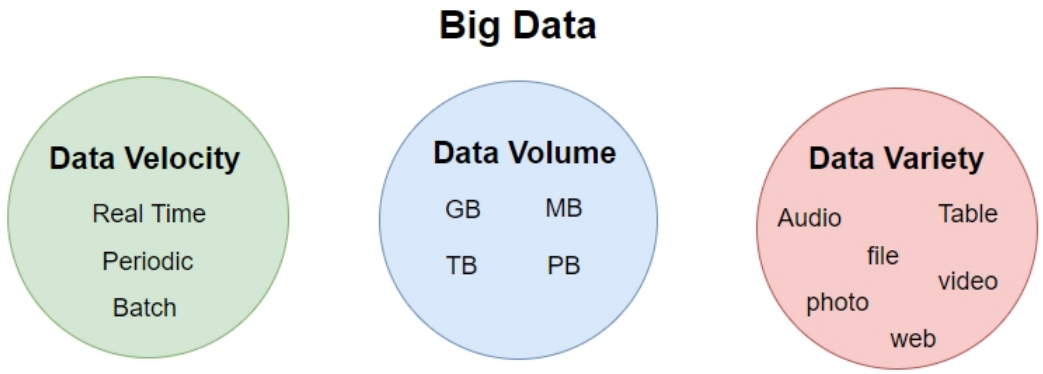
Dari rumusan masalah di atas maka tujuan dari penelitian adalah sebagai berikut:

1. Mempelajari cara kerja algoritma *Agglomerative Clustering* berbasis MapReduce untuk mereduksi *big data*.
2. Mengkustomisasi dan mengimplementasikan algoritma *Agglomerative Clustering* pada lingkungan Spark.
3. Melakukan eksperimen pada lingkungan sistem tersebar Spark untuk mengukur kinerja algoritma lingkungan Spark.
4. Membuat modul program yang dapat memudahkan pengguna menginterpretasikan data yang telah direduksi.

5 Detail Perkembangan Pengerjaan Skripsi

Detail bagian pekerjaan skripsi sesuai dengan rencan kerja/laporan perkembangan terkahir :

1. **Melakukan studi literature mengenai *big data***
   Status : baru ditambahkan pada semester ini
   Hasil : *Big data* adalah istilah yang menggambarkan kumpulan data dalam jumlah yang sangat besar, baik data yang terstruktur maupun data yang tidak terstruktur. Data-data tersebut menyimpan informasi yang bisa dianalisis dan diproses untuk memberikan wawasan kepada organisasi atau perusahaan. Data-data tersebut dihasilkan dari satu atau lebih sumber dengan kecepatan yang tinggi dan format yang berbeda-beda. Karena ukuran dan keberagaman data, *big data* menjadi sulit untuk ditangani atau diproses jika hanya menggunakan manajemen basis data atau aplikasi pemrosesan data traditional.



Gambar 1: Gambar *big data volume, velocity, variety*

Berdasakan gambar diatas (Gambar **??**), *big data* memiliki tiga karakteristik diantaranya:

(a) *Volume*: *big data* memiliki jumlah data yang sangat besar sehingga dalam proses pengolahan data dibutuhkan suatu penyimpanan yang besar dan dibutuhkan analisis yang lebih spesifik.

(b) *Velocity*: *big data* memiliki aliran data yang sangat cepat.

(c) *Variety*: *big data* memiliki bentuk format data yang beragam baik terstruktur ataupun tidak terstruktur dan bergantung pada banyaknya sumber data.

Big data sangat bermanfaat ketika diterapkan di berbagai macam bidang seperti bisnis, kesehatan, pemerintahan, pertanian dan lainya. Ketika organisasi mampu menggabungkan jumlah data besar yang dimilikinya dengan analisis bertenaga tinggi, organisasi dapat menyelesaikan tantangan dan masalah yang berhubungan dengan bisnis seperti:

(a) Menentukan akar penyebab kegagalan untuk setiap masalah bisnis.

(b) Menghasilkan informasi mengenai titik penting penjualan berdasarkan kebiasaan pelanggan dalam membeli.

(c) Menghitung kembali seluruh risiko yang ada dalam waktu yang singkat.

(d) Mendeteksi perilaku penipuan yang dapat mempengaruhi organisasi.

2. **Melakukan studi literatur mengenai Hadoop**
Status : Ada sejak rencana kerja skripsi
Hasil : Hadoop dikembangkan oleh Doug Cutting dan Mike Cafarella pada tahun 2005 yang saat itu bekerja di Yahoo. Nama Hadoop berdasarkan mainan 'Gajah' anak dari Doug Cutting. Hadoop adalah sebuah framework atau platform open source berbasis Java. Hadoop memiliki kemampuan untuk penyimpanan dan memproses data dengan skala yang besar secara terdistribusi pada *cluster* yang terdiri dari perangkat keras komoditas. Hadoop menggunakan teknologi Google MapReduce dan Google File System (GFS) sebagai fondasinya [**?**]. Beberapa karakteristik yang dimilki Hadoop adalah sebagai berikut:

(a) *Open Source*: Hadoop merupakan proyek *open source* dan kodenya bisa dimodifikasi sesuai kebutuhan.

(b) *Distributed computing*: Data disimpan secara terdistribusi pada HDFS di berbagai *cluster* dan data diproses secara parallel pada node-node di *cluster*.

(c) *Fast*: Hadoop sangat baik untuk melakukan *high-volume batch processing* karena kemampuannya melakukannya secara parallel.

(d) *Fault Tolerance*: Hadoop melakukan duplikasi data di beberapa node yang berbeda. Ketika sebuah node gagal memproses data, node yang memiliki duplikat data bisa mengantikanya untuk memproses data tersebut.

(e) *Reliability*: Kegagalan mesin bukan masalah bagi Hadoop karena adanya duplikasi data.

(f) *High availability*: Data dapat diambil dari sumber yang lain meskipun kegagaln mesin karena adanya duplikasi data.

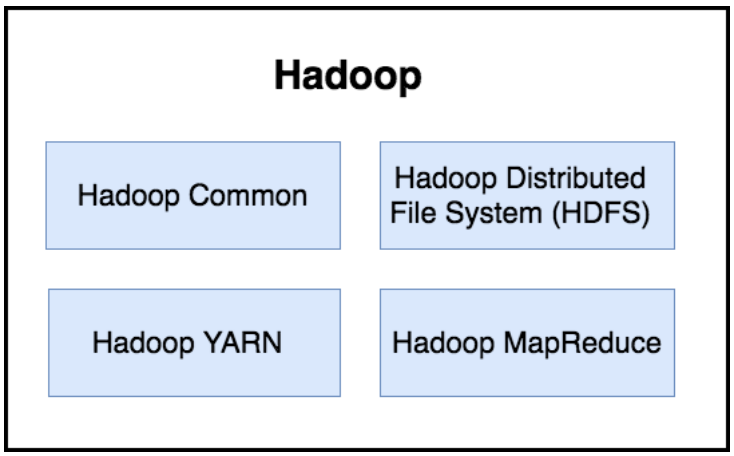
(g) *Scalability*: Hadoop dengan sangat mudah dapat menambahkan node yang lebih banyak kedalam *cluster*.

(h) *Flexibility*: Hadoop dapat menangani data terstruktur maupun data tidak terstruktur.

(i) *Economic and cost effective*: Hadoop tidak terlalu mahal karena berjalan pada *cluster* terdiri dari perangkat keras komoditas.

(j) *Easy to use*: Hadoop mempermudah pengguna dalam merancang program parallel. Hadoop sudah menangani hal-hal terakit *distributed computing*.

(k) *Data locality*: Algoritma MapReduce akan didekatkan kepada *cluter* dan tidak sebaliknya. Ukuran data yang besar lebih sulit untuk dipindahkan dibanding ukuran algoritma yang kecil.



Gambar 2: Gambar modul-modul Hadoop

Bedasarkan gambar diatas (Gambar **??**), *framework* Apache Hadoop terdiri dari beberapa modul. Module-module tersebut membentuk dan membantu untuk memproses data yang bersakala besar. Modul-modul itu diantaranya adalah:

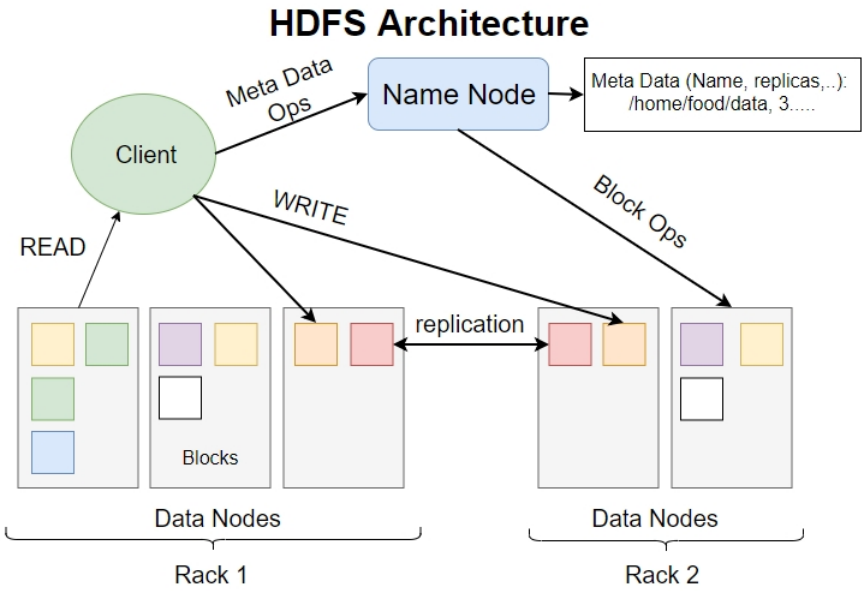
(a) Hadoop Common, module ini mengandung *library* dan *tools* yang dibutuhkan module Hadoop lainnya.

(b) Hadoop Distributed File System (HDFS), sebuah *file-system* terdistribusi milik Hadoop untuk penyimpanan data.

(c) Hadoop YARN, *resource-management platform* yang bertanggung jawab untuk mengatur sumber daya pada *cluster*.

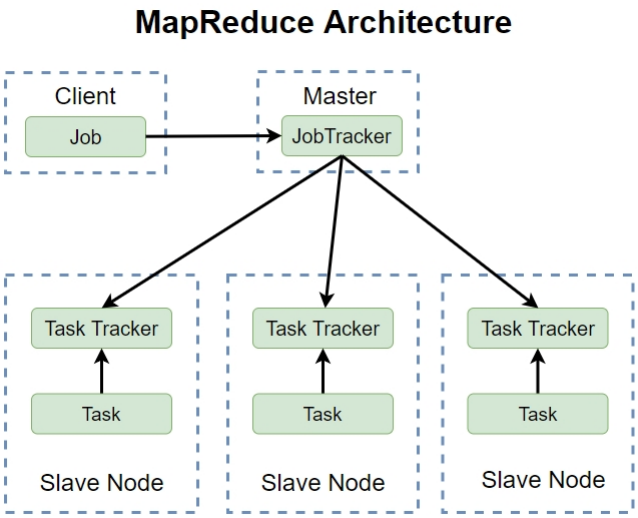
(d) MapReduce, sebuah programming model untuk pemrosesan skala besar.

Master Slave Architecture Pada Hadoop



Gambar 3: Gambar arsitektur HDFS

Hadoop meimplementasikan *Master Slave Architecture* pada komponen primernya yaitu HDFS dan MapReduce. Bedasarkan (Gambar **??**), NameNode atau disebut master node bertugas mengatur *file system namepsace* seperti *open, close, rename* file dan direktori. Selain itu, NameNode meregulasi akses user terhadap file dan mengatur block mana yang akan diolah oleh DataNode. DataNode atau bisa disebut *slave node* merupakan pekerja dari HDFS. DataNode bertangungjawab untuk membaca dan menulis request dari *file system* Hadoop. NameNode bisa melakukan block *create, delete,* dan *replicate* ketika diberi instruksi dari *master node*.



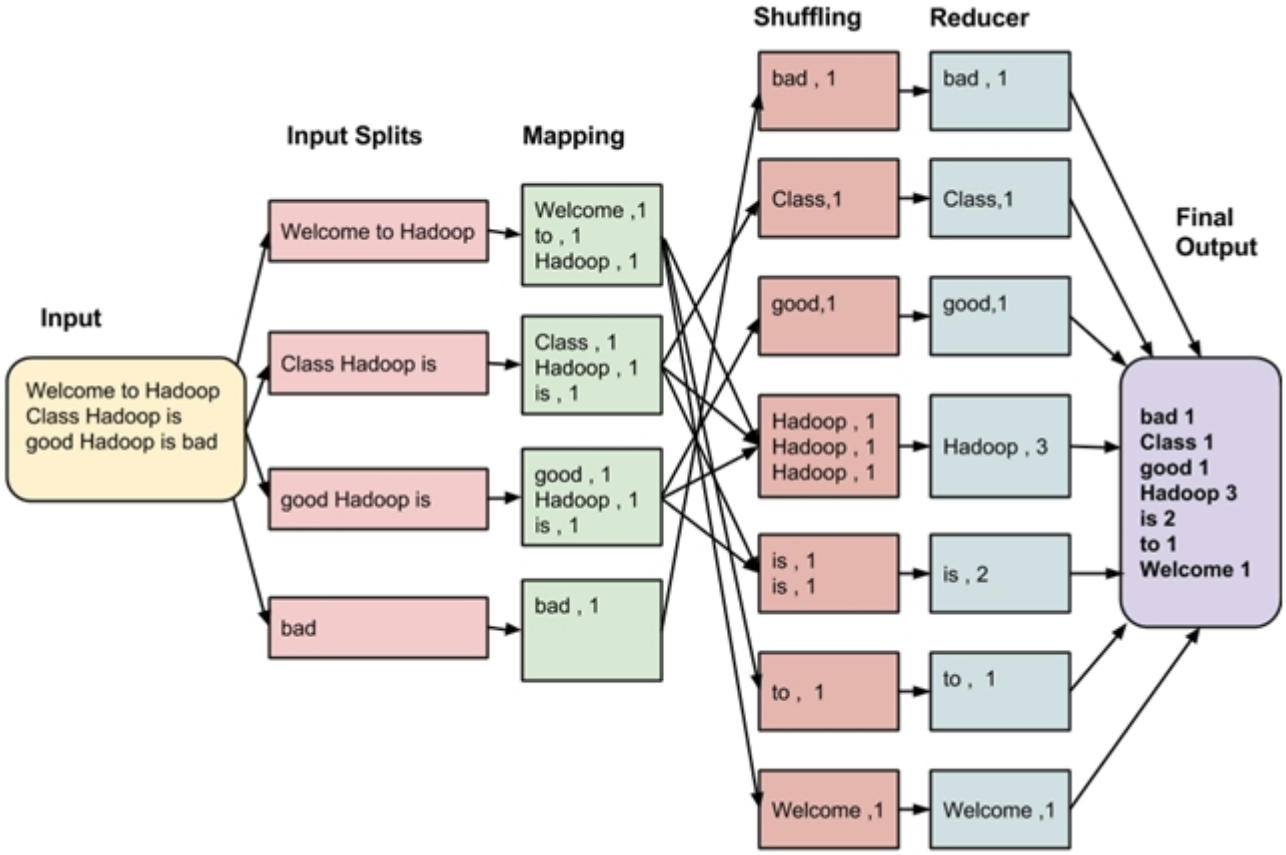
Gambar 4: Gambar arsitektur MapReduce

Bisa dilihat pada (Gambar **??**) yaitu arsitektur MapReduce, *master node* disebut JobTracker dan *slave node* disebut TaskTracker. JobTracker adalah jembatan antara user dan fungsi map maupun reduce. Ketika Map atau Reduce job diterima oleh JobTracker, job tersebut akan dimasukan kedalam queue dan menjalankan job sesuai urutan it first-come/first-served. Job yang dieksekusi akan ditugaskan

kepada TaskTracker oleh JobTracker. TaskTracker atau *slave* menproses *task* yang diberikan oleh JobTracker. Pergerakan data dari fase Map ke fase Reducer ditangani oleh TaskTracker.

MapReduce

MapReduce adalah model pemrograman untuk memproses data berukuran raksasa secara terdistribusi dan paralel dalam cluster yang terdiri atas ribuan komputer. Dalam memproses data, secara garis besar MapReduce dapat dibagi dalam dua proses yaitu proses Map dan proses Reduce. Setiap fase memiliki *key-value pairs* sebagai input dan output [?]. Kedua jenis proses ini didistribusikan atau dibagi-bagikan ke setiap komputer dalam suatu cluster dan berjalan secara paralel tanpa saling bergantung satu dengan yang lainnya. Proses Map bertugas untuk mengumpulkan informasi dari potongan-potongan data yang terdistribusi dalam tiap komputer dalam cluster. Hasilnya diserahkan kepada proses Reduce untuk diproses lebih lanjut. Hasil proses Reduce merupakan hasil akhir.



Gambar 5: Gambar proses MapReduce

Bedasarkan (Gambar **??**) , berikut adalah langkah-langkah proses awal *input* sampai akhir dari MapReduce:

(a) *Input* dibagi menjadi *input split* yang berukuran sama, untuk setiap input splits dibuatlah Map *task*.

(b) Pada fase Map, data pada setiap *split* akan dihitung berapa banyak kemunculan kata tersebut dan dijadikan pair <word, frequency> sebagai *ouput*.

(c) Setelah phase Mapping, output dari fase ini memasuki tahap Shuffling dimana tugasnya adalah

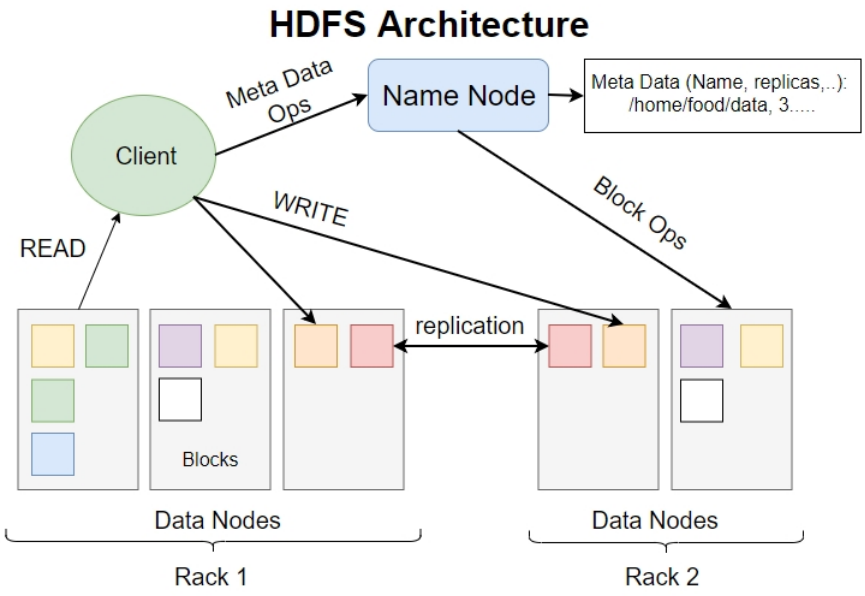
untuk mengkonsolidasikan *records* yang relevant dari *output* fase Map. Dalam contoh ini, kata-kata yang sama disatukan bersama dengan frekuensi masing-masing.

(d) Terakhir adalah fase Reduce dimana *ouput* dari *shuffling* akan dikumpulkan. Nilai-nilai dari fase *shuffling* akan digabungkan menjadi sebuah *output*. *Output* akan disimpan pada HDFS.

Hadoop Distributed File System

Hadoop Distributed File System (HDFS) adalah sistem file terdistribusi yang dirancang untuk berjalan pada perangkat keras komoditas. HDFS berbeda dari sistem file terdistribusi lainnya adalah karena sifat *fault tolerant* yang tinggi dan dirancang untuk digunakan pada perangkat keras biasa. HDFS menyediakan akses throughput yang tinggi ke data aplikasi dan cocok untuk aplikasi yang memiliki set data yang besar. HDFS awalnya dibangun sebagai infrastruktur untuk proyek mesin pencari web Apache Nutch.

Kegagalan perangkat keras sudah biasa terjadi. HDFS mungkin terdiri dari ratusan atau ribuan mesin server, masing-masing menyimpan bagian dari data *file sustem*. Faktanya, ada sejumlah besar komponen dan setiap komponen memiliki probabilitas kegagalan. Hal ini menandakan bahwa beberapa komponen HDFS selalu tidak berfungsi. Oleh karena itu, deteksi kesalahan dan pemulihan otomatis yang cepat dari sistem adalah tujuan arsitektur inti dari HDFS.



Gambar 6: Gambar Hadoop Distributed File System

HDFS dirancang untuk menyimpan file yang berukuran sangat besar di seluruh mesin dalam *cluster* yang besar. HDFS menyimpan setiap file sebagai blok yang berurutan. semua blok dalam file kecuali blok terakhir memiliki ukuran yang sama. Bisa dilihat pada (Gambar **??**) bahwa blok-blok file direplikasi untuk memiliki *fault tolerance* yang tinggi. Ukuran blok dan banyaknya replika dapat dikonfigurasi untuk setiap file. Faktor replikasi dapat ditentukan pada waktu pembuatan file dan dapat diubah nantinya. File dalam HDFS ditulis sekali dan hanya memiliki satu penulis setiap saat.

Namenode mengelola *file system namespaces*. Tidak hanya itu, tugas lain yang dimiliki NameNode adalah memelihara *file system tree* dan metadata untuk semua file dan direktori di pohon. NameNode membuat semua keputusan terkait replikasi blok. NameNode secara berkala menerima Heartbeat dan Blockreport dari masing-masing DataNode di *cluster*. Heartbeat mengimplikasikan bahwa DataNode berfungsi dengan benar. Blockreport berisi daftar semua blok pada DataNode. DataNode merupkan pekerja dari sistem file Hadoop. DataNode menyimpan dan mengambil blok ketika diperintahkan oleh NameNode. Selain itu, DataNode melaporkan daftar blok-blok yang disimpan kepada NameNode secara rutin.

3. **Melakukan studei literatur mengenai Spark)**
   Status : Ada sejak rencana kerja skripsi
   Hasil :

Pembahasan Umum Spark

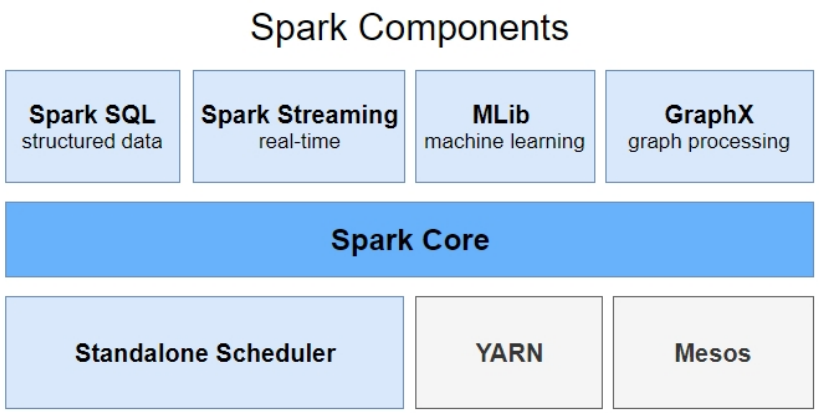
Apache spark adalah sebuah *cluster computing platform* dirancang untuk kecepatan dan *general-purpose*. Spark dirancang bedasarkan model MapReduce yang populer untuk memberikan dukungan yang efisien kepada banyak tipe komputasi, termasuk *interactive query*, dan *stream processing* [**?**]. Kecepatan merupakan kunci dalam memproses data set yang besar, perbedaan waktu dalam eksplorasi data bisa dari beberapa menit sampai beberapa jam tergantung pada kecepatan. Salah satu fitur utama Spark yang ditawarkan adalah kemmapuannya untuk melakukan *in memory computations*. Selain itu, sistem Spark lebih efisien daripada MapReduce dalam menjalankan applikasi yang rumit pada disk.

Pada sisi *general-purpose*, Spark dirancang untuk mencakup berbagai beban kerja yang sebelumnya diperlukan sistem terdistribusi terpisah, termasuk aplikasi *batch*, *iterative algorithms*, *interactive query*, dan *streaming*. Dengan mendukung beban kerja tersebut di mesin yang sama, Spark membuat pekerjaan lebih mudah dan murah untuk menggabungkan pemrosesan yang berbeda jenis. Dengan begitu, Spark mengurangi beban dalam *maintaining* tools-tools yang terpisah.

Spark di desain agar sangat *accessible* dengan memberikan API sederhana untuk Python, Java, Scala, dan SQL. Spark denganmudah berintegrasi dengan tools *Big Data* lainnya, terutama Hadoop. Spark bisa berjalan dalam Hadoop *cluster* dan mengakses sumber data Hadoop mana saja.

Komponen Spark

Proyek Spark memiliki beberapa komponen yang terintegrasi dengan erat. Sebagai *core*, Spark adalah "mesin komputasi" yang bertanggung jawab untuk penjadwalan, distribusi, dan pemantauan aplikasi yang terdiri dari banyak *computational task* tersebar di banyak pekerja, mesin, atau *computing cluster*. Karena *core engine* dari Spark cepat dan *general-purpose*, Spark menjalankan banyak *higer-level components* untuk menangani berbagai macam pekerjaan khuses seprti SQL atau *machine learning*. Komponen-komponen ini dirancang untuk saling beroperrasi dengan erat, Spark membiarkan Anda menggabungkan komponen seperti *library* dalam suatu proyek perangkat lunak.



Gambar 7: Gambar komponen pada Spark

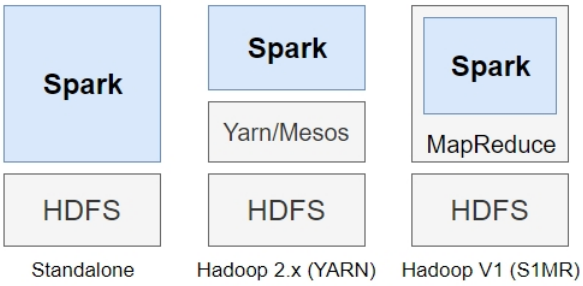
Bedasarkan (Gambar **??**), Spark memiliki beberapa komponen sebagai berikut:

- Spark Core: Spark Core berisi fungsi-fungsi dasar Spark, termasuk komponen untuk tugas penjadwalan, manajemen memori, pemulihan kesalahan, berinteraksi dengan sistem penyimpanan, dan banyak lagi. Spark Core merupakan rumah bagi API *resilient distributed datasets*(RDD), yang merupakan abstraksi pemrograman utama Spark. RDD mewakili suatu koleksi *item* yang didistribusikan di banyak node komputasi yang dapat dimanipulasi secara parallel. Spark Core menyediakan banyak API untuk membangun dan memanipulasi ini koleksi.
- Spark SQL: Spark SQL adalah sebuah *package* untuk bekerja dengan data yang terstuktur. *Package* ini memungkinkan melakukan kueri pada data terstrukut melalui SQL serta varian Apache Hive dari SQL disebut Hive Query Language (HQL) dan *pacakage* ini mendukung banyak sumber data, termasuk tabel Hive, Parket, dan JSON. Selain menyediakan antarmuka SQL untuk Spark, Spark SQL memungkinkan *developer* untuk memadukan kueri SQL dengan manipulasi data terprogram yang didukung oleh RDD pada Python, Java, dan Scala, semua dalam satu aplikasi, sehingga menggabungkan SQL dengan analitik yang rumit. Integrasi ketat dengan lingkungan komputasi yang kaya disediakan oleh Spark membuat Spark SQL tidak seperti gudang data open source lainnya.
- Spark Streaming: Spark Streaming adalah komponen Spark yang memungkinkan pemrosesan data dari *live streaming*. Contoh *data steam* termasuk file log yang dihasilkan oleh server web produksi, atau antrian pesan yang berisi pembaruan status yang diposting oleh pengguna layanan web. Spark Streaming menyediakan API yang mirip dengan Spark CoreâĂŹs RDD API untuk memanipulasi aliran data. Hal ini membuat *developer* mudah memplajari proyek dan berpindah antar aplikasi yang memanipulasi data yang disimpan dalam memori, pada disk, atau tiba dalam *real time*. Di balik API-nya, Spark Streaming dirancang untuk menyediakan tingkat toleransi kesalahan, throughput, dan skalabilitas yang sama seperti Spark Core.
- MLlib: Spark hadir dengan *library* yang berisi fungsi pembelajaran mesin secara umum (ML), *library* ini disebut MLlib. MLlib menyediakan beberapa jenis algoritma pembelajaran mesin, termasuk klasifikasi, regresi, pengelompokan, dan penyaringan kolaboratif, serta pendukung fungsionalitas seperti *model evaluation* dan *data import*. MLlib juga menyediakan beberapa *lower-level* ML *primitives*, termasuk *generic gradient descent optimization algorithm*.
- GraphX: GraphX adalah sebuah *library* untuk memanipulasi grafik dan melakukan *graph-parallel computations*. Seperti Spark Streaming dan Spark SQL, GraphX âĂŃâĂŃmemperluas API Spark RDD, memungkinkan kita untuk membuat *directed graph* dengan *arbitrary propertiesi* yang melekat pada setiap *vertex* dan *edge*. GraphX âĂŃâĂŃjuga menyediakan berbagai operator untuk

memanipulasi grafik dan memiliki library yang penuh dengan *graph algorithms* yang umum seperti PageRank dan *triangle counting*.

- Cluster Managers: Spark dirancang untuk *scale up* secara efisien dari satu hingga ribuan node komputasi. Untuk mencapai hal ini dan memaksimalkan fleksibilitas, Spark dapat menjalankan lebih dari satu variasi manajer klaster seperti Hadoop YARN, Apache Mesos, *simple cluster manager* pada diri Spark sendiri yang disebut *Standalone Scheduler*.

Tiga Cara Membangun Spark di Atas Hadoop



Gambar 8: Gambar beberapa cara instalasi Spark

Ada tiga cara untuk meinstalasi Spark bedasarkan (Gambar **??**) diatas, ketiga cara tersebut akan dijelaskan dibawah:

- *Standalone*: Spark *standalone* berarti Spark menempati tempat di atas HDFS (Hadoop Distributed File System) dan ruang dialokasikan untuk HDFS, secara eksplisit. Di sini, Spark dan MapReduce akan berjalan berdampingan untuk mencakup semua pekerjaan percikan di cluster.
- Hadoop Yarn: Spark berjalan pada Yarn tanpa perlu pra-instalasi atau akses root. Cara ini membantu mengintegrasikan Spark ke dalam ekosistem Hadoop atau Hadoop stack. Cara ini memungkinkan komponen lain untuk berjalan di atas tumpukan.
- Spark pada MapReduce: Spark pada MapReduce digunakan untuk menjalankan job-job pada spark selain untuk *standalone deployment*. Dengan adanya SIMR, pengguna dapat memulai Spark dan menggunakan *Spark Shell* tanpa akses administratif.

Resilient Distributed Datasets (RDD)

Resilient Distributed Datasets (RDD)adalah struktur data dasar Spark. RDD adalah koleksi benda-benda yang didistribusikan secara permanen. Setiap dataset dalam RDD dibagi menjadi beberapa partisi yang dapat dikomputasi pada node yang berbeda pada *cluster* [**?**]. RDD dapat berisi jenis objek Python, Java, atau Scala, termasuk kelas yang ditentukan pengguna. Spark memanfaatkan konsep RDD untuk mencapai operasi MapReduce yang lebih cepat dan efisien.

Secara umun, RDD adalah kumpulan *read-only, partitioned collection* dari *records*. RDD dapat dibuat melalui operasi deterministik dari data pada penyimpanan yang stabil atau RDD lainnya. RDD adalah kumpulan elemen *fault tolerance* yang dapat dioperasikan secara paralel.

Data sharing pada MapReduce lebih lambat dibanding RDD karena replikasi, serialisasi, dan disk IO. Sebagian besar aplikasi Hadoop menghabiskan lebih dari 90 persen waktunya untuk melakukan operasi *read-write* keapda HDFS.

Untuk menangani masalah tersebut, dibuatlah *framework* khusus disebut Apache Spark. Ide utama dari Spark adalah RDD, Spark mendukung *in-memory computation*. Spark menyimpan status memori sebagai objek di seluruh pekerjaan dan objek dapat dibagi diantara *jobs*. *Data sharing* dalam memori lebih cepat 10 hingga 100 kali lipat dibanding *network* atau *disk*.

Berikut adalah sifat-sifat dari RDD :

- *In Memory*: RDD merupakan sebuah kumpulan object yang berada pada memori. Meskipun RDD memiliki pilihan untuk disimpan di memori, hardisk, atau bahkan keduanya, kecepatan eksekusi Spark berdasarkan fakta bahwa data berada dalam memori. Bukan mengambil dari hard disk setiap kali menjalankan operasi.
- *Partitioned*: RDD melakukan partisi adalah suatu syarat mutlak teknik untuk mengoptimalkan performa pada distributed system. Hal ini bertujuan meminimalisir network traffic dan sekaligus pemungkas high performance workloads. Dalam melakukan partisi, data yang berupa key/value ditempatkan sesuai dengan rentang key yang bernilai sama. Tujuannya untuk meminimalisir data yang berpindah-pindah.
- *Typed*: Data didalam RDD akan selalu digolongkan berdasarkan tipe data.
- *Lazy evaluation*: *Transformation* pada Spark bersifat âĂİlazyâĂİ atau malas. Ini artinya data didalam RDD tidak akan tersedia sampai dilakukan sebuah action.
- *Immutable*: RDD yang telah dibuat tidak dapat berubah. Meskipun demikian, RDD dapat ditransformasi menjadi sebuah RDD baru dengan melakukan perintah *transformation* pada RDD.
- *Parallel*: RDD dapat dioperasikan secara pararel.
- *Cacheable*: Karena RDD bersifat *lazy evaluation*, setiap action yang dilakukan pada RDD akan menyebabkan RDD mengevaluasi kembali transformasi yang menyebabkan pembuatan RDD. Karena hal ini merupakan sifat yang berdampak buruk pada dataset berukuran besar, maka Spark memberikan pilihan untuk melakukan cache data di memory ataupun pada hard disk.

Ada dua cara untuk membuat sebuah RDD. Cara pertama adalah dengan memuat dataset eksternal. Sedangkan cara alternatif adalah dengan mendistribustikan sebuah koleksi objek seperti *list* atau *set*. Ketika sebuah RDD telah dibuat, ada dua tipe operasi yaitu *transformations* dan *actions*. *Transformations* membuat RDD baru dari RDD sebelumnya. Berbeda dengan *tranformations*, *actions* mengembalikan nilai hasil komputasi bedasarkan RDD. Hasil dari *actions* akan dikembalikan kepada *driver program* atau di simpan pada penyimpanan eksternal seperti HDFS.

```
//Contoh code membuat RDD dari sumber eksternal
val lines = sc.textFile("/path/to/README.md")

//Contoh memparalelkan sebuah koleksi pada driver program
val lines = sc.parallelize(["pandas", "ilikepandas"])
```

Transformations pada RDD adalah sebuah operasi yang menerima RDD sebagi masukan dan mengembalikan satu atau lebih RDD baru. RDD masukan tidak berubah karena sifat RDD adalah *immutable* yang berarti tidak bisa diubah ketika dibuat. *Transformations* bersifat *lazy*, *transformation* tidak langsung dieksekusi, melainkan Spark akan mencatat *tranfomartion* apa saja yang dilakukan pada RDD awal. *Transformations* akan dieksekusi ketika sebuah *actions* dipanggil.

Berikut adalah contoh filter() transformation di Scala:

```
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line => line.contains("error"))
```

Berikut adalah Tabel ?? berisi daftar *transformations* yang umum pada Spark:

Tabel 1: Tabel transformastions

Transformations	Penjelasan
map(func)	Mengembalikan dataset terdistribusi baru yang dibentuk dengan melewatkan setiap elemen melalui fungsi func.
filter(func)	Mengembalikan dataset baru yang dibentuk dengan memilih elemen-elemen yang mengembalikan nilai true dari fungsi func.
flatMap(func)	Mirip dengan *map*, tetapi setiap *item input* dapat dipetakan ke nol atau lebih *item output*.
mapPartitions(func)	Mirip dengan *map*, tetapi berjalan secara terpisah pada setiap partisi (blok) dari RDD, jadi func harus bertipe Iterator <T> => Iterator <U> ketika menjalankan pada RDD tipe T
mapPartitionsWithIndex(func)	Mirip dengan mapPartitions, tetapi harus menyediakan func dengan nilai integer yang mewakili indeks partisi, jadi func harus bertipe (Int, Iterator <T>) => Iterator <U> ketika menjalankan pada RDD tipe T.
sample(withReplacement, fraction, seed)	Mengambil sebagian data sebagai data dengan atau tanpa penggantian menggunakan *random number generator seed* yang diberikan.
union(otherDataset)	Mengembalikan *dataset* baru yang mengandung *element* dari sumber dan *dataset*lainnya.
intersection(otherDataset)	Mengembalikan *dataset* baru yang berisi potongan *element* dari sumber dan *argument*.
distinct([numPartitions])	Mengembalikan *dataset* baru yang mengandung *element* yang unik dari *dataset* sumber.
groupByKey([numPartitions])	Mengembalikan *dataset* baru bertipe *pairs* (K, Iterable<V>) dari sumber *dataset* bertipe (K, V).
groupByKey(func,[numPartitions])	Mengembalikan *dataset* baru bertipe *pairs* (K, V) yang sudah diagregasi bedasarkan *key* dan *reduce function* yang diberikan.
sortByKey([ascending], [numPartitions])	Mengembalikan *dataset* baru berupa *pairs* (K, V) yang terurut secara menaik atau menurun badsarkan parameter boolean yang diberikan.
join(otherDataset, [numPartitions])	Mengembalikan gabungan *dataset* berupa *pairs* (K, V) dan (K, W) menjadi *pairs* (K, (V,W)).
cogroup(otherDataset, [numPartitions])	Mengembalikan *dataset* berupa *tuples* (K, (Iterable<V>, Iterable<W>)) dari *pairs* (K, V) dan (K, W).
cartesian(otherDataset)	Mengembalikan *dataset* berupa *paris* (T, U) dari *dataset* T dan U.

Berikut adalah contoh operasi RDD:

```
val lines = sc.textFile("data.txt")
val pairs = lines.map(s => (s, 1))
val counts = pairs.reduceByKey((a, b) => a + b)
```

Actions merupakah operasi yang mengembalikan sebuah nilai kepada *driver program* atau tempat penyimpanan eksternal. Untuk mengembalikan sebuah nilai, kita bisa menggunakan take(), count(), collect(), dan *actions* lainya. Operasi take() digunakan untuk mengambil sebagian kecil *element* pada RDD. Ketika menggunakan collect(), memori pada satu komputer harus cukup untuk menampung

seluruh *dataset*. Operasi tersebut baiknya digunakan pada *dataset* yang berukuran kecil dan *dataset* berukuran besar bisa disimpan pada tempat penyimpanan eksternal. Setiap kali *actions* baru dipanggil, seluruh RDD akan dikomputasi dari akarnya. Untuk mencapai efisiensi yang lebih tinggi, bisa dilakukan *persist* terhadap *intermediate results*.

Berikut adalah Tabel **??** berisi daftar *actions* yang umum pada Spark:

Tabel 2: Tabel Actions

Actions	Penjelasan
reduce(func)	Mengagregasikan seluruh *element* pada *dataset* menggunakan fungsi yang diberikan pada *parameter*.
collect()	Mengembalikan seluruh *dataset* sebagai array kepada *driver program*.
count()	Mengembalikan jumlah *element* pada *dataset*.
first()	Mengembalikan *element* pertama pada *dataset*.
take(n)	Mengembalikan sebuah array dengan n jumlah *element* pertama dari *dataset*.
takeSample(withReplacement, x, [seed])	Mengembalikan sebuah array dengan x jumlah *element* secara acak dari *dataset*.
takeOrdered(n, [ordering])	Mengembalikan sebuah array dengan n jumlah *element* pertama dari *dataset* secara terurut.
saveAsTextFile(path)	Menyimpan *dataset* sebagai *text file* pada direktori yang ditentukan.
saveAsSequenceFile(path)	Menyimpan *dataset* sebagai Hadoop SequenceFile pada direktori yang ditentukan.
saveAsObjectFile(path)	Menyimpan *dataset* sebagai format yang sederhana menggunakan Java Serialization pada direktori yang ditentukan.
countByKey()	Menjumlahkan *pairs* (K, V) bedasarkan *key* dan mengembalikan sebuah *pairs* berisi (K, int).
foreach(func)	Memproses setiap *element* pada *dataset* menggunakan fungsi func yang diberikan.

4. **Melakukan studi literatur mengenai bahasa pemrograman Scala**
   Status : Ada sejak rencana kerja skripsi
   Hasil : Scala adalah sebuah bahasa pemrograman yang diciptakan oleh Martin Odersky yaitu seorang Profesor di Ecole Polytechnique Federale de Lausanne, sebuah kampus di Lausanne, Swiss. Kata Scala sendiri merupakan kependekan dari âĂIJScalable LanguageâĂİ. Karena Scala berjalan diatas Java Virtual Machine (JVM), Scala memiliki performa yang relatif cepat dan juga memungkinkan untuk menggabungkan kode di Scala dengan di Java. Termasuk library, framework dan tool yang ada di Java, bisa gunakan di Scala. Scala menggabungkan konsep Object Oriented Programming (OOP) yang dikenal di Java dengan konsep Functional Programming (FP). Adanya konsep FP inilah yang menjadikan Scala sangat ekspresif, nyaman dan menyenangkan untuk digunakan.

   Perintah *scalac* digunakan untuk mengkompilasi program Scala dan akan menghasilkan beberapa file kelas di direktori saat ini. Salah satunya akan disebut file .class. Ini adalah bytecode yang akan berjalan di Java Virtual Machine (JVM) dengan menggunakan perintah *scala*.

   Expressions

   suatu *ekspressions* adalah pernyataan atau argumen yang dapat dikomputasi.

```
1 + 1
2 + 2
```

Ekspressions dapat dikembalikan dengan perintah println.

```
println(1)
println(100) // 100
println(1 + 1) // 2
println("Hi!") // Hi!
```

Ekspressions atau pernyataan seperti diatas dapat disimpan dalam sebuah variable ada dua jenis variable di Scala yaitu val dan var. Setelah val diinisialisasi, val tidak dapat diisi kembali artinya isi dari val tidak dapat diubah.

```
val x = 2 + 5
val x = 10 //tidak akan di compile
val y = 7
val coba:Int = 200
```

variable mirip dengan value, tetapi nilai variable bisa di isi kembali.

```
var x = 2 + 2
x = 4
println(x) // 4
x = 7
println(x) // 7
```

Secara eksplisit kita bisa menyatakan tipe dari sebuah var atau val dengan cara:

```
var x: Int = 1 + 1 // Int merupakan tipe dari variable x
val y: Long = 987654321 // Long merupakan tipe dari variable y
val z: Char = 'a' // Char merupakan tipe dari variable z
```

Blocks

Block digunakan untuk menggabungkan *expressions*. Berikut adalah contoh *block*:

```
println({
    val x = 1 + 1
    x + 1
}) // 3
```

Loop dan Conditional

loop adalah struktur pengulangan yang memungkinkan untuk menulis secara efisien suatu loop yang perlu dieksekusi sejumlah kali. Ada berbagai bentuk loop dalam Scala yang dijelaskan di bawah ini:

```
for( var x <- Range ){
    statement(s);
}
```

```
var x = 0
while (x < 10) {
    println(x)
    i += 1
}
```

Percabangan adalah pengujian sebuah kondisi. Jika kondisi yang diuji tersebut terpenuhi, maka program akan menjalankan pernyataan-pernyataan tertentu. Jika kondisi yang diuji salah, program akan menjalankan pernyataan yang lain. Berikut adalah contoh percabangan dalam bahasa Scala:

```
if( x < 20 ){
    println("This is if statement");
}

if( x < 20 ){
    if( x< 5) {
        println("smallest");
    }
}

if( x < 10 ){
    println("This is bigger");
} else {
    println("This is smaller");
}

if( x == 1 ){
    println("1");
} else if (x == 2){
    println("2");
}
```

Functions

Functions adalah *expression* yang mempunyai atau menerima parameter. Sebuah function yang tidak memliki nama disebut anonymous function. Berikut adalah contoh anonymous function dan function biasa. Sebuah function bisa memiliki lebih dari satu parameter.

```
(x: Int) = > x + 1 // Anonymous function

val addOne = (x: Int) => x + 1 // function biasa
println(addOne(2)) // 3

val add = (x: Int, y: Int) => x + y
println(add(1, 2)) // 3
```

Pada sisi sebelah kiri tanda => adalah parameter-parameter sebuah function, pada sisi kanan merupakan expresi-expresi yang melibatkan parameter tersebut.

Methods

Method hampir sama dengan functionn, tetapi method memiliki beberapa perbedaan. Method harus di definisikan dengan kata kunci def, diikuti dengan nama method, paramter-parameter dari method tersebut, tipe kembalian method, dan *body dari method tersebut.*

```
def add(x: Int, y: Int): Int = x + y
println(add(1, 2)) // 3
```

Method bisa mempunyai lebih dari satu parameter *list*

```
def addThenMultiply(x: Int, y: Int)(multiplier: Int): Int = (x + y) * multiplier
println(addThenMultiply(1, 2)(3)) // 9
```

Method yang tidak memiliki parameter

```
def name: String = System.getProperty("user.name")
println("Hello, " + name + "!")
```

Method berbeda dengan functions bisa memiliki *multi-line expressions*

```
def getSquareString(input: Double): String = {
    val square = input * input
    square.toString
}
```

Expresi terakhir dari method menjadi nilai yang akan dikembalikan. Scale memunyai *keyword* return, tetapi sangat jarang digunakan.

Class dan Object

Class pada Scala di definisikan dengan kata kunci class diikuti dengan namanya dan terakhir adalah consturctor parameter.

```
class Greeter(prefix: String, suffix: String) {
    def greet(name: String): Unit = {
        println(prefix + name + suffix)
    }
}
```

Dibawah adalah cara mendeklarasi objek dan pemanggilan method pada Scala

```
val greeter = new Greeter("Hello, ", "!")
greeter.greet("Scala developer")
```

Objek dapat dianggap sebagai sesuatu instansi yang tunggal pada class itu sendiri. Untuk mendefinisikan objek, digunakan kata kunci Object.

```
object IdFactory {
    private var counter = 0

    def create(): Int = {
      counter += 1
      counter
    }
}

val newId: Int = IdFactory.create()
println(newId) // 1
val newerId: Int = IdFactory.create()
println(newerId) // 2
```

Main method adalah sebuah pintu masuk dari sebuah program. Java Virtual Machine membutuhkan sebuah main method yang dinamakan main dan menerima satu argument, seubah array bertipe string.

Menggunakan *object*, kita bisa mendefinisikan sebuah main method seperti berikut:

```
object Main {
    def main(args: Array[String]): Unit = {
        println("Hello, Scala developer!")
    }
}
```

Higher Order Function

Pada bahasa Scala ada yang disebut sebagai Higher Order Function. Higher Order Function merupakan sebuah function yang menerima function lainya sebagai parameter dan mengembalikan sebuah function sebagai hasilnya. Berikut adalah contoh-contoh Higher Order Function:

```
val salaries = Seq(20000, 70000, 40000)
val doubleSalary = (x: Int) => x * 2
val newSalaries = salaries.map(doubleSalary) // List(40000, 140000, 80000)
```

Kita bisa mempersingkat kode dengan sebuah function anonymous dan langsung dimasukan pada parameter

```
val salaries = Seq(20000, 70000, 40000)
val newSalaries = salaries.map(x => x * 2) // List(40000, 140000, 80000)
```

Kita juga dapat memasukan method pada parameter higher order function, compiler Scala akan mengubah sebuah method menjadi function.

```
case class WeeklyWeatherForecast(temperatures: Seq[Double]) {

    private def convertCtoF(temp: Double) = temp * 1.8 + 32 // sebuah method

    def forecastInFahrenheit: Seq[Double] = temperatures.map(convertCtoF) }
    // method sebagai parameter
}
```

Salah satu alasan untuk menggunakan higher order function adalah untuk mengurangi kode yang berlebihan. Katakanlah ada beberapa metode yang dapat menaikkan gaji seseorang dengan berbagai faktor. Tanpa membuat fungsi urutan tinggi, mungkin terlihat seperti ini:

```
object SalaryRaiser {

    def smallPromotion(salaries: List[Double]): List[Double] =
      salaries.map(salary => salary * 1.1)

    def greatPromotion(salaries: List[Double]): List[Double] =
      salaries.map(salary => salary * math.log(salary))

    def hugePromotion(salaries: List[Double]): List[Double] =
      salaries.map(salary => salary * salary)
}
```

Perhatikan bahwa masing-masing dari ketiga method hanya berbeda pada faktor perkalian. Untuk menyederhanakan, kita dapat mengeluarkan kode yang redundan menjadi higher order function seperti:

```
object SalaryRaiser {

    //Higher Order Function
    private def promotion(sal: List[Double], func: Double => Double): List[Double] =
      salaries.map(func)

    def smallPromotion(sal: List[Double]): List[Double] =
      promotion(sal, salary => salary * 1.1)

    def bigPromotion(sal: List[Double]): List[Double] =
      promotion(sal, salary => salary * math.log(salary))

    def hugePromotion(sal: List[Double]): List[Double] =
      promotion(sal, salary => salary * salary)
}
```

5. **Melakukan studi literatur mengenai teknik-teknik reduksi data dan algoritma agglomerative**
   Status : Ada sejak rencana kerja skripsi
   Hasil : Meledaknya ukuran data, jumlag *record*, dan attribut data membuat pengenbangan baru dalam teknik mereduksi data. Mereduksi data merupakan bagian penting dalam bidang *big data*. Data direkduksi agar lebih mudah diolah dan diterapkan teknik-teknik *data mining*. Teknik yang digunakan dalam mereduksi data sangat tergantung dengan tipe data yang ditangani dan tujuan akhir dalam mereduksi data. Dalam mereduksi data, data yang direduksi harus bisa mewakili data metah secara keseluruhan. Bila data yang sudah direduksi tidak mewakili data awal, tidak ada gunanya untuk mereduksi data.

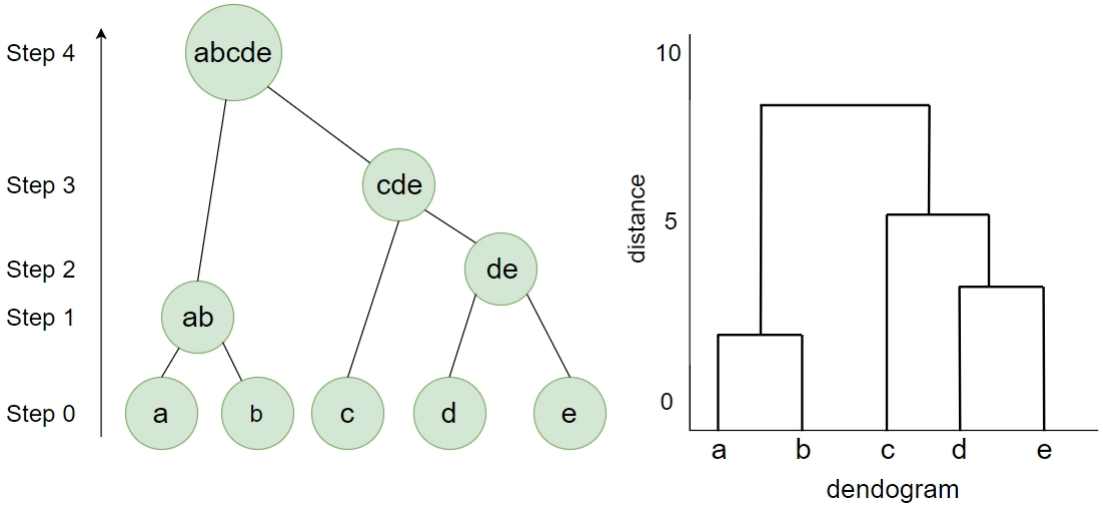
   Berikut adalah contoh dan penjelasan beberapa teknik:

- *Missing Values Ratio*: Kolom data dengan nilai yang terlalu banyak hilang kemungkinan tidak akan membawa banyak informasi yang berguna. Kolom data dengan jumlah nilai yang hilang lebih besar dari batas yang diberikan dapat dihapus. Semakin tinggi batas, semakin agresif pengurangannya.
- *Low Variance Filter*: Mirip dengan teknik sebelumnya, kolom data dengan sedikit perubahan dalam data membawa sedikit informasi. Jadi semua kolom data dengan varians lebih rendah dari batas yang diberikan akan dihapus. Normalisasi diperlukan sebelum menerapkan teknik ini.
- *High Correlation Filter*: kolom data dengan kecenderungan yang sangat mirip biasanya membawa informasi yang sangat mirip. Dalam hal ini, hanya satu saja yang cukup untuk diberikan kepada model pembelajaran mesin. Di sini kita menghitung koefisien korelasi antara kolom numerik dan antara kolom nominal sebagai Pearson's Product Moment Coefficient dan nilai chi square Pearson masing-masing. Pasangan kolom dengan koefisien korelasi lebih tinggi dari batas digabungkan menjadi hanya satu.
- *Random Forests / Ensemble Trees*: Algoritma ini digunakan pada klasifikasi data dalam jumlah yang besar. Klasifikasi *random forest* dilakukan melalui penggabungan pohon dengan melakukan *training* pada sampel data yang dimiliki. Penggunaan pohon yang semakin banyak akan mempengaruhi akurasi yang akan didapatkan menjadi lebih baik. Penentuan klasifikasi dengan *random forest* diambil berdasarkan hasil voting dari pohon yang terbentuk. Pemenang dari pohon yang terbentuk ditentukan dengan vote terbanyak. Pembangunan pohon pada *random forest* sampai dengan mencapai ukuran maksimum dari pohon data. Akan tetapi, pembangunan pohon *random forest* tidak dilakukan *pruning* yaitu sebuah metode untuk mengurangi kompleksitas ruang. Pembangunan dilakukan dengan penerapan metode seleksi fitur secara acak untuk meminimalisir kesalahan. Pembentukan pohon dengan sample data menggunakan variable yang diambil secara acak dan menjalankan klasifikasi pada semua tree yang terbentuk. *Random forest* menggunakan *Decision Tree* untuk melakukan proses seleksi. Pohon yang dibangun dibagi secara rekursif dari data pada kelas yang sama. Pemecahan digunakan untuk membagi data berdasarkan jenis atribut yang digunakan.
- *Principal Component Analysis (PCA)*: PCA adalah prosedur statistik yang secara ortogonal mengubah koordinat n asli dari satu set data menjadi satu set baru dari n koordinat yang disebut komponen utama. Sebagai hasil dari transformasi, komponen utama pertama memiliki kemungkinan varian terbesar; setiap komponen yang berhasil memiliki kemungkinan varian tertinggi di bawah batasan yang bersifat ortogonal terhadap. PCA menjaga hanya komponen m $<$ n komponen pertama mengurangi dimensi data sambil mempertahankan sebagian besar informasi data, yaitu variasi dalam data.
- *Backward Feature Elimination*: Pada teknik ini, pada iterasi yang diberikan, algoritma klasifikasi yang dipilih dilatih pada n fitur input. Kemudian, dihapus satu fitur masukan pada satu waktu dan melatih model yang sama pada fitur masukan n-1 kali n. Fitur input yang penghapusannya telah menghasilkan peningkatan terkecil dalam tingkat kesalahan dihapus, meninggalkan kita dengan fitur input n-1. Klasifikasi ini kemudian diulang menggunakan fitur n-2, dan seterusnya. Setiap iterasi k menghasilkan model yang dilatih pada fitur n-k dan tingkat kesalahan e (k). Memilih tingkat kesalahan maksimum yang dapat ditolerir, kami menetapkan jumlah terkecil fitur yang diperlukan untuk mencapai kinerja klasifikasi dengan algoritme pembelajaran mesin yang dipilih.
- *Forward Feature Construction*: Teknik ini adalah teknik kebalikan dari *Backward Feature Elimination*. Dimulai dari satu fiture, dan secara bertahap ditambahkannya fitur yang menghasilkan kenaikan performa yang paling tinggi. Kedua algoritma memiliki konsumsi waktu dan kompu-

tasi yang tinggi. Algoritma tersebut baiknya digunakan pada *data set* dengan colom input yang sedikit.

Hierarchical Clustering Algorithm (HCA) adalah metode analisis kelompok yang berusaha untuk membangun sebuah hirarki dengan mengelompokan data. Dengan mengelompokan data-data tersebut, data pada kelompok yang sama memiliki kemiripan yang tinggi dan data pada kelompok yang berbeda memiliki kemiripan yang rendah. Dalam reduksi data, *cluster* yang merepresentasikan data-data pada *cluster* tersebut akan diggunakan untuk mengganti data-data mentah. Seberapa efektif cara ini tergantung dengan sifat data yang ditangani. Data yang bisa dikelompokan kedalam cluster yang berbeda akan sangat cocok dengan cara ini. Pada dasarnya HCA dibagi menjadi dua jenis yaitu Agglomerative (Bottom-Up) dan Devisive (Top-Down).

Agglomeratice hierarchical clustering memulai dengan setiap objek membentuk sub-cluster tersendiri dan secara iterative menggabung sub-cluster menjadi sub-cluster yang lebih besar sampai semua objek berada pada satu cluster. Pada tahap pegabungan, cluster yang terdekat bedasarkan similaritas tertentu akan digabungkan menjadi satu. Pada setiap iterasi akan ada 2 buah cluster yang dibagung menjadi satu, maka dari itu agglomerative clustering akan hanya membutuhkan maksimal n iterasi. Dendogram sangat umum digunakan untuk menggambarkan proses HCA.



Gambar 9: Gambar *dendogram*

Berdasakan gambar diatas (Gambar **??**), berikut adalah langkah-langkah:

(a) Hitung Matrik Jarak antar objek.

(b) Gabungkan dua kelompok terdekat berdasarkan parameter kedekatan yang ditentukan.

(c) Perbarui Matrik Jarak antar objek untuk merepresentasikan kedekatan diantara kelompok baru dan kelompok yang masih tersisa.

(d) Ulangi langkah 2 dan 3 higga hanya satu kelompok yang tersisa.

Berikut adalah tiga metode pengelompokan Agglomerative Hierarchical:

- Single Linkage (Jarak Terdekat): Sebuah sub-tree atau sub-cluster cp dikelompokan dengan sub-cluster cq menggunakan minimum distance antara *object members* pada cp dan cq (edge terdekat)

```
dmin = min(dij)
1 <= i <= m
1 <= j <= n

dij = adalah jarak antara objek i pada cp dan j pada cq;
m = jumlah objek pada sub-cluster cp
n = jumlah objek pada sub-cluster cg.
```

- Complete Linkage (Jarak Terjauh): Sebuah sub-tree atau sub-cluster cp dikelompokan dengan sub-cluster cq menggunakan maximum distance antara *object members* pada cp dan cq (edge terjauh)

```
dmin = max(dij)
1 <= i <= m
1 <= j <= n

dij = adalah jarak antara objek i pada cp dan j pada cq;
m = jumlah objek pada sub-cluster cp
n = jumlah objek pada sub-cluster cg.
```

- Average Linkage (Jarak Rata-Rata): Sebuah sub-tree atau sub-cluster cp dikelompokan dengan sub-cluster cq menggunakan average distance antara *centroid ci* dan *centroid cg*.

```
dmeans = distance(centroidcp, centroidcq)
centroidcp = jarak rata-rata semua artibute pada sub-cluster Cp
centroidcp = jarak rata-rata semua artibute pada sub-cluster Cg
```

6. **Melakukan instalasi Spark dan eksplorasi Spark shell**
   Status : Ada sejak rencana kerja skripsi
   Hasil :

   Pada bagian ini, akan dijelaskan tahap-tahap untuk melakukan instalasi Apache Spark. Apache Spark yang akan digunakan adalah Apache Spark versi x Spark dapat berjalan diatas berbagai sistem operasi seperti Window Windows dan UNIX systems (Contoh Linux, Mac OS). Sebelum memulai instalasi Apache Spark, ada beberapa kebutuhan yang harus dipenuhi seperti instalasi JAVA dan Scala. Berikut adalah langkah-langkah untuk memastikan kita telah memenuhi kebutuhan minimal:

   - Cek apakah Java telah diinstall dan versi java yang diinstall adalah setidaknya 8+ karena Spark berjalan pada versi minimal Java 8+. Berikut adalah command untuk memastikan java telah terinstall.

   ```
   java -version

   Java(TM) SE Runtime Environment (build 1.8.0_112-b15)
   ```

   - Jika scala belum di installasi pastikan di install dengan versi minimal 2.11.x.

   ```
   scala -version

   Scala code runner version 2.11.6 -- Copyright 2002-2013, LAMP/EPFL
   ```

Bila kita tidak memliki JAVA dan Scala pada komputer, berikut adalah cara menginstall JAVA dan Scala untuk kebutuhan Spark:

- Berikut adalah comand-comand untuk menginstall Java menggunakan terminal pada sistem operasi Linux:

```
sudo apt-get update
sudo apt-get install default-jdk
```

- Berikut adalah comand-comand untuk menginstall Scala menggunakan terminal pada sistem operasi Linux:

```
sudo apt-get update
sudo apt-get install scala
```

Instalasi dapat dilakukan ketika syarat-syarat diatas telah dipenuhi. Berikut adalah langkah-langkah instalsi Apache Spark:

(a) Pertama, donwload versi spark yang diinginkan dari link https://spark.apache.org/downloads.html

(b) Kemudian extract Spark tar dengan command berikut:

```
tar xvf spark-2.3.1-bin-hadoop2.7.tgz

su âĂŞ
Password:

cd /home/usr/Downloads/
mv spark-2.3.1-bin-hadoop2.7 /usr/local/spark
exit
```

(c) Kemudian kita harus menkonfigurasi environment variable untuk Spark. Ubah file .bsshrc dengan menbahkan command berikut pada file.

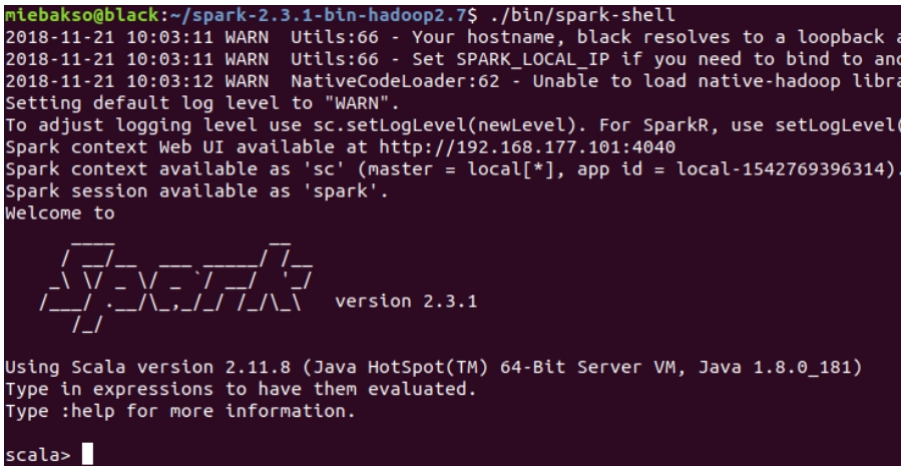
```
export PATH=$PATH:/usr/local/spark/bin
```

(d) Terakhir jalankan command berikut untuk memastikan perubahan telah terjadi pada file /.bashrc.

```
source ~/.bashrc
```

(e) Ketika spark diinstall dengan benar, maka dengan command spark-shell kita bisa menjalankan spark shell (Gambar **??**).

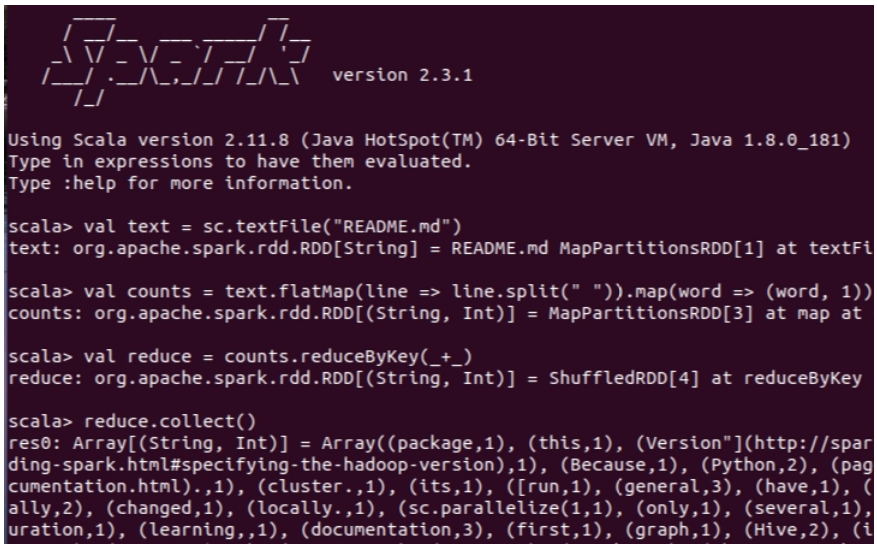
```
$ SPARK_HOME/bin/spark-shell
```



Gambar 10: Gambar *Spark Shell*

Eksplorasi Spark Shell

Pada bagian ini, penulis akan menjelaskan percoba *word count* pada file text README.md. Penulis akan menggunakan Spark Shell untuk menjalankan perintah-perintah agar spark bisa menghitung jumlah setiap kata yang ada pada text file tersebut. Setiap kata yang sama akan dijumlahkan. Pada bagian ini akan digunakan tranformation dan juga action.



Gambar 11: Gambar *Word Count*

Bedasarkan gambar diatas (Gambar **??**), berikut adalah langkah-langkah percobaan *word count*:

(a) Pertama, kita jalankan spark shell dengan command berikut pada terminal.

```
$ ./bin/spark-shell
```

(b) Setelah itu, kita akan membuat text RDD dengan megambil sumber eksternal yaitu file README.md. Command dibawah digunakan untuk membuat RDD dari file eksternal.

```
val text = sc.textFile("README.md")
text: org.apache.spark.rdd.RDD[String] = README.md MapPartititonsRDD[1] at textFile at <consol
```

(c) Kemudian, kita akan memisahkan setiap kata menjadi pairs dengan kata sebaya key dan 1 sebagi value setiap kata.

```
val counts = text.textflatMap(line => line.split(" ")).map(word => (word, 1))
counts: org.apache.spark.rdd.RDD[(String, int)] = ShuffledRDD[3] at map at <console>:25
```

(d) Langkah selanjutnya, kita akan menghitung jumlah setiap kata dengan cara berikut.

```
val reduce = counts.reduceByKey(_+_)
reduce: org.apache.spark.rdd.RDD[(String, int)] = ShuffledRDD[4] at reduceByKey at <console>
```

(e) Terakhir, kita akan mengambil hasil.

```
reduce.collect()
res0: Array[(String, Int)] = Array((package,1), (Python,2), .....
```

Instalasi Spark pada multi-node cluster

Pada bagian ini, penulis akan menjelaskan langkah-langkah yang diperlukan untuk menginstall Spark pada multi-node cluster. Berikut adalah langkah-langkah yang harus dilakukan:

(a) Tambahkan entri dalam file host (master dan slave)

```
$ sudo vim /etc/hosts

\\tambahkan IP master dan juga slave

<MASTER-IP> master
<SLAVE1-IP> slave1
<SLAVE2-IP> slave2
<SLAVE3-IP> slave3
```

(b) Kemudian install Java pada setiap master dan slave, jangan lupa cek versi java yang diinstall

```
$ sudo apt-get update
$ sudo apt-get install default-jdk

\\Setelah itu cek versi java yang diinstal

$ java -version
```

(c) Kemudian install Scala pada setiap master dan slave, jangan lupa cek versi scala yang diinstall.

```
$ sudo apt-get update
$ sudo apt-get install scala

\\Setelah itu cek scala java yang diinstal

$ scala -version
```

(d) Install Open SSH Server-Client pada master.

```
\\Untuk install Open SSH
$ sudo apt-get install openssh-server openssh-client

\\Generate key pairs
$ ssh-keygen -t rsa -P ""

Selanjutkna kita perlu konfigurasi SSH pada slave dan juga master.
Copy .ssh/id_rsa.pub miliki master kepada .ssh/authorized_keys untuk master dan juga slave.
```

(e) Setelah itu, kita akan download dan install Spark pada setiap slave dan master.

```
\\Dowload versi spark yang diinginkan pada
https://spark.apache.org/downloads.html

\\extract spark dengan comand berikut
$ tar xvf spark-2.3.0-bin-hadoop2.7.tgz

\\pindahkan spark kepada direktori yang diinginkan
$ sudo mv spark-2.3.0-bin-hadoop2.7 /direktori/yang/diinginkan/
```

(f) Sesudah download maka kita harus mengubah file bashrc.

```
$ sudo vim ~/.bashrc

export PATH = $PATH:/usr/local/spark/bin

$ source ~/.bashrc
```

(g) Kemudian kita harus konfigurasi Spark master dengan mengubah file spark-env.sh.

```
$ cd /usr/local/spark/conf
$ cp spark-env.sh.template spark-env.sh

\\Edit spark-env.sh
$ sudo vim spark-env.sh

export SPARK_MASTER_HOST='<MASTER-IP>'
export JAVA_HOME=<Path_of_JAVA_installation>

Kemudian edit file slave pada /usr/local/spark/conf

$ sudo vim slaves

master
slave1
slave2
slave3
```

(h) Sekarang, kita bisa menjalankan spark cluster

```
$ cd /usr/local/spark
$ ./sbin/start-all.sh
```

```
//Untuk memberhentikan cluster masukan comand berikut
$ ./sbin/start-all.sh
```

7. **Merancang sebagian kustomisasi algoritma berbasis spark**
   Status : Ada sejak rencana kerja skripsi
   Hasil : Berikut adalah rancangan algoritma kustomisasi yang masih belum sempurna.

```
object Hello extends App {

        var x:CustomAgglo = new CustomAggloe()
}

class CustomAgglo(var rec: Map[Int,List[Int]], max:Int, var dist:Int, var coff:Int)

    //create array list
    var treeList = Array.fill[SingleTree](max)
    //Immutable MASALAH, tidak ada arraylist
    var count:Int
    var isProceed:Boolean

    def customRun(): Unit ={
        count = 0;
        for( x <- rec){
            //Form independent tree from x add x to treeList

            isProceed=false
            if(count==max){
                //form dendogram tree using dist type
                //form object cluster using distace measure
                //clear object tree & remove from memory
                isProceed=true
            }
        }
        if(isProceed==false){
            //form dendogram tree using dist type
            //form object cluster using distace measure
            //clear object tree & remove from memory
        }
    }

}

class SingleTree(var attVal: List[Int], var clsLabel:Int){
    var x = 2

}
```

6 Pencapaian Rencana Kerja

Langkah-langkah kerja yang berhasil diselesaikan dalam Skripsi 1 ini adalah sebagai berikut:

1. Melakukan studi literatur *big data*.
2. Melakukan studi literatur Hadoop.
3. Melakukan studi literatur Spark.
4. Melakukan studi literatur bahasa pemrograman Scala.
5. Melakukan studi literatur algoritma reduksi data dan agglomerative.
6. Membuat rancangan algoritma reduksi data dalam bahasa Scala.
7. Mengerjakan dokumen skripsi sampai bab 4 awal.

7 Kendala yang Dihadapi

Kendala - kendala yang dihadapi selama mengerjakan skripsi :

- Kesulitan dalam mempelajari bahasa Scala.
- Terlalu banyak les bahasa Inggris dan bahasa Jerman.
- Komputer lab skripsi ke lock terus menerus kita ingin mencoba dan harus bergantian dengan yang lain.

Bandung, 20/11/2018

Matthew Ariel Wangsit

Menyetujui,

Nama: Veronica Sri Moertini
Pembimbing Tunggal