

SKRIPSI

**REDUKSI BIG DATA DENGAN ALGORITMA AGGLOMERATIVE
CLUSTERING UNTUK SPARK**



Matthew Ariel

NPM: 2015730010

**PROGRAM STUDI TEKNIK INFORMATIKA
FAKULTAS TEKNOLOGI INFORMASI DAN SAINS
UNIVERSITAS KATOLIK PARAHYANGAN
2018**

UNDERGRADUATE THESIS

**BIG DATA REDUCTION WITH AGGLOMERATIVE CLUSTERING
ALGORITHM FOR SPARK**



Matthew Ariel

NPM: 2015730010

**DEPARTMENT OF INFORMATICS
FACULTY OF INFORMATION TECHNOLOGY AND SCIENCES
PARAHYANGAN CATHOLIC UNIVERSITY
2018**

ABSTRAK

Big data perlu direduksi untuk menghemat tempat penyimpanan. Algoritma *Hierarchical Agglomerative Clustering* dapat digunakan untuk mereduksi data. Dengan bantuan sistem terdistribusi seperti Hadoop, proses reduksi data dapat dilakukan secara paralel dan lebih cepat. Sayangnya, teknologi Hadoop masih dapat dikatakan 'terlalu lambat' dalam melakukan proses reduksi data karena hasil sementara dari setiap tahap akan disimpan di disk sampai dibutuhkan kembali di tahap selanjutnya.

Untuk mempercepat proses reduksi data, Hadoop dapat digantikan dengan Spark. Spark adalah sistem terdistribusi, mirip seperti Hadoop. Tetapi, yang membedakan antara Hadoop dengan Spark adalah pada cara penyimpanan sementara saat melakukan proses reduksi data. Hadoop menggunakan disk sebagai tempat penyimpanan sementara, sedangkan Spark menggunakan memori sebagai tempat penyimpanan sementara. Pembacaan dan penulisan akan lebih cepat saat menggunakan memori dibandingkan dengan menggunakan disk, sehingga Spark akan lebih cepat dibandingkan dengan Hadoop.

Perangkat lunak dibuat untuk mengimplementasi algoritma *Hierarchical Agglomerative Clustering* dalam Spark. Pengujian juga dilakukan dengan membandingkan waktu eksekusi algoritma *Hierarchical Agglomerative Clustering* saat diimplementasikan pada Hadoop dan saat diimplementasikan pada Spark. Waktu eksekusi dicatat untuk ukuran data yang berbeda-beda.

Berdasarkan hasil pengujian, Spark memiliki waktu eksekusi yang lebih cepat dibandingkan dengan Hadoop. Spark memiliki waktu eksekusi yang lebih cepat dibandingkan dengan Hadoop karena data disimpan dan dibaca pada memori.

Kata-kata kunci: Reduksi Data, *Hierarchical Agglomerative Clustering*, Spark

ABSTRACT

Big data need to be reduce to save storage space. The Hierarchical Agglomerative Clustering algorithm can be used to reduce data. With the help of distributed systems such as Hadoop, reduction process can be done in parallel with less execution time. Unfortunately, Hadoop technology can still be said to be 'too slow' in the process of data reduction because temporary results from each stage will be stored on the disk until it is needed again at a later stage.

To speed up the data reduction process, Hadoop can be replaced with Spark. Spark is a distributed system, similar to Hadoop. However, what distinguishes Hadoop from Spark is the way to temporarily store data in reduction processes. Hadoop uses the disk as its temporary storage, while Spark uses memory as its temporary storage. Reading and writing process will be faster when using memory than using disks, Spark will be faster than Hadoop.

The Hierarchical Agglomerative Clustering algorithm is implemented in the software. Experiment were done by comparing the execution time of the Hierarchical Agglomerative Clustering algorithm when implemented on Hadoop and on Spark. The execution time is recorded for different data sizes.

Based on the experiment, Spark has a faster execution time compared to Hadoop. Spark has a faster execution time compared to Hadoop because data is stored and read from memory.

Keywords: Data Reduction, *Hierarchical Agglomerative Clustering*, Spark

DAFTAR ISI

DAFTAR ISI	ix
DAFTAR GAMBAR	xi
DAFTAR TABEL	xiii
1 PENDAHULUAN	1
1.1 Latar Belakang	1
1.2 Rumusan Masalah	2
1.3 Tujuan	2
1.4 Batasan Masalah	2
1.5 Metodologi	3
1.6 Sistematika Pembahasan	3
2 LANDASAN TEORI	5
2.1 Big Data	5
2.2 Algoritma Hierarchical Clustering	6
2.3 Hadoop	12
2.3.1 Penjelasan Hadoop	12
2.3.2 Hadoop Distributed File System (HDFS)	14
2.3.3 MapReduce	15
2.3.4 YARN	17
2.4 Spark	18
2.4.1 Pembahasan Umum Spark	18
2.4.2 Komponen Spark	19
2.4.3 Tiga Cara Membangun Spark di Atas Hadoop	20
2.4.4 Arsitektur Spark	21
2.4.5 Resilient Distributed Datasets (RDD)	22
2.5 Scala	25
2.5.1 Expressions	25
2.5.2 Blocks	26
2.5.3 Loop dan Conditional	26
2.5.4 Functions	27
2.5.5 Methods	28
2.5.6 Class dan Object	28
2.5.7 Higher Order Function	29
3 STUDI DAN EKSPLORASI APACHE SPARK	33
3.1 Instalasi Apache Spark	33
3.2 Eksplorasi Spark Shell	35
3.3 Instalasi Apache Spark pada multi-node cluster	36
3.4 Percobaan Spark Submit	38

4	ANALISIS DAN PERANCANGAN	45
4.1	Analisis Masalah	45
4.1.1	Identifikasi Masalah	45
4.1.2	Analisis Hierarchical Agglomerative Clustering MapReduce	46
4.1.3	Analisis Masukan dan Keluaran	48
4.1.4	Diagram Alur	49
4.1.5	Analisis Hierarchical Agglomerative Clustering untuk Spark	51
4.2	Perancangan Perangkat Lunak	57
4.2.1	Diagram Use Case dan Skenario	57
4.2.2	Diagram Kelas	59
4.2.3	Rancangan Antarmuka	63
5	IMPLEMENTASI DAN PENGUJIAN PERANGKAT LUNAK	67
5.1	Implementasi Perangkat Lunak	67
5.1.1	Lingkungan Perangkat Keras	67
5.1.2	Lingkungan Perangkat Lunak	67
5.1.3	User Interface	68
5.2	Pengujian Fungsional Perangkat Lunak	71
5.3	Hasil Eksperimen Perangkat Lunak	73
5.4	Percobaan Dampak Partisi pada Performa Perangkat Lunak Spark dan Hadoop	74
6	KESIMPULAN DAN SARAN	103
6.1	Kesimpulan	103
6.2	Saran	103
	DAFTAR REFERENSI	105
A	KODE PROGRAM	107
B	KODE PROGRAM UNTUK ANTARMUKA	113

DAFTAR GAMBAR

2.1	Karakteristik <i>big data</i>	5
2.2	Matriks jarak	6
2.3	Matriks jarak	7
2.4	<i>dendrogram</i>	7
2.5	Metode <i>single linkage</i>	8
2.6	Metode <i>complete linkage</i>	8
2.7	Metode <i>centroid linkage</i>	9
2.8	Matriks jarak	9
2.9	Hasil penggabungan <i>cluster</i>	10
2.10	Hasil rekalkulasi	11
2.11	Hasil akhir <i>dendrogram</i>	11
2.12	Perpotongan <i>dendrogram</i>	12
2.13	Modul-modul Hadoop	13
2.14	Arsitektur HDFS	14
2.15	Arsitektur MapReduce	16
2.16	Proses MapReduce	17
2.17	Proses menjalankan aplikasi pada YARN	18
2.18	Komponen pada Spark	19
2.19	Macam-macam cara instalasi Spark	20
2.20	Arsitektur Spark	21
3.1	<i>Spark Shell</i>	34
3.2	<i>Word Count</i>	35
3.3	IntelliJ IDEA	39
3.4	Proyek sbt	40
3.5	Konfigurasi proyek	41
3.6	Struktur proyek	41
3.7	Konfigurasi sbt	42
3.8	<i>object</i> WordCount	42
3.9	Kode WordCount	42
3.10	JAR	43
3.11	Hasil perintah 'sbt package'	43
3.12	Penggumpulan JAR kepada <i>spark-submit</i>	43
3.13	Alamat Spark UI	44
3.14	Spark UI	44
4.1	Penulisan kepada disk di MapReduce	46
4.2	Penulisan kepada memori di Spark	46
4.3	Diagram alur perangkat lunak	49
4.4	Partisi RDD	50
4.5	RDD <i>parsing</i> dan kelas <i>Node</i>	50
4.6	<i>Worker</i> memproses partisi	50
4.7	Proses reduksi pada <i>reducer</i> dan kelas <i>Pattern</i>	51

4.8	Penyimpanan pola pada HDFS	51
4.9	Contoh perhitungan matriks dan pembentukan dendrogram	56
4.10	Contoh pemotongan <i>dendrogram</i>	57
4.11	Diagram <i>use case</i> perangkat lunak <i>Hierarchical Agglomerative Clustering</i>	58
4.12	Diagram kelas	59
4.13	Kelas Main, SparkConfig, SparkContext	59
4.14	Kelas DataReducer	60
4.15	Kelas Dendrogram	60
4.16	Kelas Cluster	61
4.17	Kelas Pattern	62
4.18	Kelas Node	62
4.19	Recangan antarmuka menu <i>submit</i>	63
4.20	Rancangan antarmuka <i>result</i>	64
4.21	Halaman web Hadoop	64
4.22	Rancangan antarmuka menu <i>Data</i>	65
4.23	Rancangan antarmuka halaman <i>list</i>	65
4.24	Rancangan antarmuka halaman data	66
4.25	Halaman web HDFS	66
5.1	Tampilan menu <i>Submit</i>	68
5.2	Tampilan menu <i>Data</i>	69
5.3	Tampilan halaman sesudah <i>submit</i>	69
5.4	Tampilan halaman <i>list</i>	70
5.5	Tampilan halaman data	71
5.6	Hasil Percobaan Partisi Spark dan Hadoop 1GB	75
5.7	Hasil Percobaan Partisi Spark dan Hadoop 2GB	76
5.8	Hasil Percobaan Partisi Spark dan Hadoop 3GB	77
5.9	Hasil Percobaan Partisi Spark dan Hadoop 5GB	78
5.10	Hasil Percobaan Partisi Spark dan Hadoop 10GB	80
5.11	Hasil Percobaan Partisi Spark dan Hadoop 15GB	81
5.12	Hasil Percobaan Partisi Spark dan Hadoop 20GB	83
5.13	Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 5 GB	84
5.14	Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 10 GB	85
5.15	Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 15 GB	86
5.16	Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 20 GB	87
5.17	Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 5 GB	88
5.18	Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 10 GB	90
5.19	Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 15 GB	91
5.20	Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 20 GB	92
5.21	Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 5 GB	93
5.22	Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 10 GB	94
5.23	Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 15 GB	95
5.24	Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 20 GB	97
5.25	Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 5 GB	98
5.26	Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 10 GB	99
5.27	Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 15 GB	100
5.28	Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 20 GB	101

DAFTAR TABEL

2.1	Tabel Data Koordinat	9
2.2	Tabel Contoh Data Cluster	12
2.3	Tabel Hasil Pola Cluster A	12
2.4	Tabel transformations	24
2.5	Tabel Actions	25
5.1	Tabel data yang digunakan pada eksperimen	74
5.2	Percobaan Jumlah Partisi Hadoop dan Spark dengan Ukuran Data 10 GB	74
5.3	Percobaan Jumlah Partisi Hadoop dan Spark dengan Ukuran Data 2 GB	75
5.4	Percobaan Jumlah Partisi Hadoop dan Spark dengan Ukuran Data 3 GB	77
5.5	Percobaan Jumlah Partisi Hadoop dan Spark dengan Ukuran Data 5 GB	78
5.6	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 10 GB	79
5.7	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 10 GB	79
5.8	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 15	81
5.9	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 15 GB	81
5.10	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 20	82
5.11	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 20 GB	82
5.12	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 5 GB	83
5.13	Percobaan Jumlah Partisi Spark dengan Ukuran Data 5 GB	84
5.14	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 10 GB	85
5.15	Percobaan Jumlah Partisi Spark dengan Ukuran Data 10 GB	85
5.16	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 15 GB	86
5.17	Percobaan Jumlah Partisi Spark dengan Ukuran Data 15 GB	86
5.18	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 20 GB	87
5.19	Percobaan Jumlah Partisi Spark dengan Ukuran Data 20 GB	87
5.20	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 5 GB	88
5.21	Percobaan Jumlah Partisi Spark dengan Ukuran Data 5 GB	88
5.22	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 10 GB	89
5.23	Percobaan Jumlah Partisi Spark dengan Ukuran Data 10 GB	89
5.24	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 15 GB	90
5.25	Percobaan Jumlah Partisi Spark dengan Ukuran Data 15 GB	91
5.26	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 20 GB	92
5.27	Percobaan Jumlah Partisi Spark dengan Ukuran Data 20 GB	92
5.28	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 5 GB	93
5.29	Percobaan Jumlah Partisi Spark dengan Ukuran Data 5 GB	93
5.30	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 10 GB	94
5.31	Percobaan Jumlah Partisi Spark dengan Ukuran Data 10 GB	94
5.32	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 15 GB	95
5.33	Percobaan Jumlah Partisi Spark dengan Ukuran Data 15 GB	95
5.34	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 20 GB	96
5.35	Percobaan Jumlah Partisi Spark dengan Ukuran Data 20 GB	96
5.36	Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 5 GB	97

5.37 Percobaan Jumlah Partisi Spark dengan Ukuran Data 5 GB	98
5.38 Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 10 GB	99
5.39 Percobaan Jumlah Partisi Spark dengan Ukuran Data 10 GB	99
5.40 Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 15 GB	100
5.41 Percobaan Jumlah Partisi Spark dengan Ukuran Data 15 GB	100
5.42 Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 20 GB	101
5.43 Percobaan Jumlah Partisi Spark dengan Ukuran Data 20 GB	101

BAB 1

PENDAHULUAN

1.1 Latar Belakang

Big data adalah sebuah istilah yang menggambarkan volume data yang besar, baik data yang terstruktur maupun data yang tidak terstruktur. Data-data tersebut memiliki potensi untuk digali menjadi informasi yang penting. Dalam bidang *big data* ada beberapa tantangan seperti volume data yang besar, kecepatan aliran data yang masuk, dan variasi data dengan format yang berbeda. Tantangan tersebut membuat aplikasi pemrosesan data tradisional tidak bisa memproses dan menganalisis *big data*. Munculah teknologi-teknologi seperti Hadoop dan Spark yang dirancang 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 pengiriman data, *disk space*, dan jumlah data yang diproses. Hasil dari reduksi *big data* harus bisa mewakili data mentahnya secara akurat.

Salah satu cara mereduksi data adalah dengan menggunakan algoritma *Hierarchical Agglomerative Clustering*. Algoritma tersebut cocok untuk data yang tidak memiliki atribut yang terlalu banyak. Journal ilmiah berjudul *Big Data Reduction Technique using Parallel Hierarchical Agglomerative Clustering* menjabarkan algoritma *Hierarchical Agglomerative Clustering* berbasis MapReduce pada Hadoop. Penelitian tersebut membuktikan bahwa data yang direduksi dengan algoritma tersebut bisa mewakili data mentah secara keseluruhan. Algoritma *Hierarchical Agglomerative Clustering* bekerja dengan mengubah setiap objek menjadi *sub-cluster*. Kemudian, *sub-cluster* akan di gabung dengan *sub-cluster* lainnya secara bertahap berdasarkan jarak antara *sub-cluster* sampai terbentuknya sebuah *cluster*. *Cluster* tersebut akan menjadi akar dari hierarki.

Meskipun hasil reduksi data dengan algoritma *Hierarchical Agglomerative Clustering* berbasis MapReduce pada Hadoop dapat mewakili data mentahnya secara akurat, MapReduce pada Hadoop memiliki kekurangan. Hadoop tidak efisien dalam melakukan proses iteratif, *intermediate data* tidak dapat disimpan pada memori. Hadoop perlu melakukan penulisan dan pembacaan kepada disk diantara setiap tahap Map dan Reduce.

Spark adalah *distributed cluster-computing framework* yang bisa menggantikan MapReduce beserta kekurangannya. *In-memory processing* pada Spark dapat mengalahkan kecepatan pemrosesan pada Hadoop MapReduce. Karena data disimpan pada RAM, kecepatan pemrosesan akan jauh lebih cepat. Spark membaca data yang akan direduksi dari RAM. Pembacaan data dari RAM akan lebih cepat dibanding disk.

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 *Hierarchical Agglomerative Clustering* yang khusus dirancang untuk lingkungan Spark. Perangkat lunak akan menampilkan hasil reduksi dalam format visual dan tabel. Dengan menggunakan Spark, waktu proses

1 reduksi data akan lebih cepat dibanding MapReduce.

2 **1.2 Rumusan Masalah**

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

- 4 1. Bagaimana cara kerja algoritma *Hierarchical Agglomerative Clustering* berbasis MapReduce untuk
5 mereduksi *big data*?
- 6 2. Bagaimana cara mengkustomisasi dan mengimplementasikan algoritma *Agglomerative Clustering*
7 pada sistem tersebar Spark?
- 8 3. Bagaimana mengukur kinerja hasil dari implementasi dari algoritma *Agglomerative Clustering* pada
9 sistem tersebar Spark?
- 10 4. Bagaimana cara mempresentasikan data yang telah direduksi?

11 **1.3 Tujuan**

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

- 13 1. Mempelajari cara kerja algoritma *Hierarchical Agglomerative Clustering* berbasis MapReduce untuk
14 mereduksi *big data*.
- 15 2. Mengkustomisasi dan mengimplementasikan algoritma *Hierarchical Agglomerative Clustering* pada
16 lingkungan Spark.
- 17 3. Melakukan eksperimen pada lingkungan sistem tersebar Spark untuk mengukur kinerja algoritma
18 lingkungan Spark.
- 19 4. Membuat modul program untuk menginterpretasikan data yang telah direduksi.

20 **1.4 Batasan Masalah**

21 Batasan masalah pada Skripsi ini adalah sebagai berikut:

- 22 1. Studi literatur Hadoop hanya dilakukan pada dasar dan file system Hadoop yaitu HDFS.
- 23 2. Studi literatur Apache Spark hanya dilakukan pada dasar, Spark RDD dan cara mengimplementasikan
24 algoritma *Hierarchical Agglomerative Clustering* pada Spark.
- 25 3. Metode reduksi data yang dibahas secara mendalam hanya metode *agglomerative clustering*.
- 26 4. Algoritma *Hierarchical Agglomerative Clustering* akan diimplementasikan secara paralel pada sistem
27 terdistribusi Spark.

1.5 Metodologi

Metodologi yang digunakan dalam pembuatan skripsi ini adalah:

1. Melakukan studi literatur tentang konsep dasar Hadoop dan sistem file Hadoop yaitu HDFS.
2. Melakukan studi literatur tentang konsep Apache Spark.
3. Melakukan studi literatur bahasa pemrograman Scala.
4. Melakukan studi literatur tentang algoritma *Hierarchical Agglomerative Clustering*.
5. Melakukan instalasi dan konfigurasi Apache Spark.
6. Melakukan eksperimen dengan bahasa pemrograman Scala.
7. Melakukan eksperimen dengan Spark RDD.
8. Melakukan kustomisasi algoritma *Hierarchical Agglomerative Clustering* untuk Spark.
9. Mencari dan mengumpulkan data uji coba yang bervolume besar.
10. Merancang dan mengimplementasikan perangkat lunak
11. Melakukan eksperimen terhadap perangkat lunak dan menganalisis hasil eksperimen.
12. Menulis dokumen skripsi.

1.6 Sistematika Pembahasan

Laporan penelitian tersusun ke dalam enam bab secara sistematis sebagai berikut:

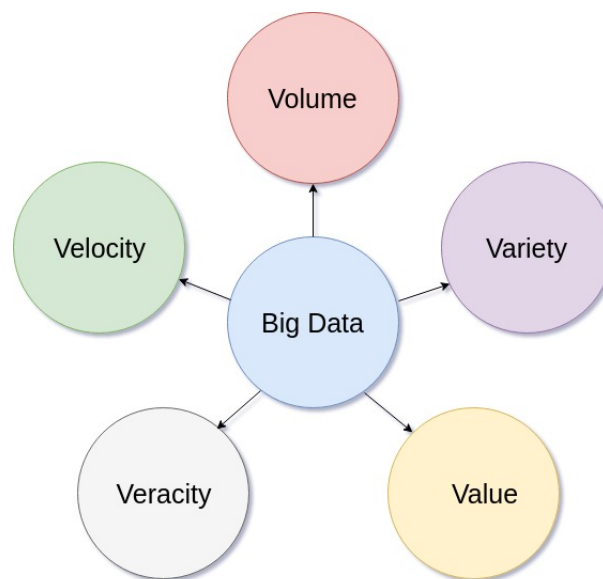
- Bab 1 Pendahuluan
Berisi latar belakang, rumusan masalah, tujuan, batasan masalah, metodologi penelitian, dan sistematika pembahasan
- Bab 2 Dasar Teori
Berisi dasar teori tentang *big data*, *Hierarchical Agglomerative Clustering*, Hadoop, Spark, dan Scala.
- Bab 3 Studi dan Eksplorasi Apache Spark
Berisi percobaan-percobaan Spark.
- Bab 4 Analisis dan Perancangan
Berisi analisis masalah, diagram alur, *use case* dan skenario, diagram kelas, dan perancangan antarmuka.
- Bab 5 Implementasi dan Pengujian
Berisi implementasi antarmuka perangkat lunak, pengujian eksperimen, dan kesimpulan dari pengujian.
- Bab 5 Implementasi dan Pengujian
Berisi kesimpulan awal sampai akhir penelitian dan saran untuk penelitian selanjutnya.

BAB 2

LANDASAN TEORI

2.1 Big Data

Big data adalah istilah yang menggambarkan kumpulan data dalam jumlah yang sangat besar, baik data yang terstruktur maupun data yang tidak terstruktur. Kumpulan data tersebut menyimpan informasi yang bisa dianalisis dan diproses untuk memberikan wawasan kepada organisasi atau perusahaan. Data-data tersebut berasal 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 tradisional [1].



Gambar 2.1: Karakteristik *big data*

Berdasarkan gambar diatas (Gambar 2.1), *big data* memiliki lima karakteristik di antaranya [1]:

1. *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.
2. *Velocity*: *big data* memiliki aliran data yang sangat cepat. Data baru dihasilkan dengan kecepatan yang tinggi dari satu atau lebih sumber.
3. *Variety*: *big data* memiliki bentuk format data yang beragam baik terstruktur ataupun tidak terstruktur dan bergantung pada banyaknya sumber data. Data dapat berupa gambar, video dan tipe data lainnya.

4. *Veracity*: *big data* mungkin mengandung data yang tidak akurat atau rusak. Kualitas data dalam *big data* bisa berbeda-beda tergantung pada sumber. Analisis *big data* akan sangat dipengaruhi dengan keakuratan data.

5. *Value*: *big data* harus memiliki *value*. Tidak ada gunanya bila kita memiliki akses terhadap *big data*, tetapi data-data tersebut tidak memiliki nilai apapun. Data yang tidak memiliki nilai adalah data yang tidak berguna dan memakan biaya untuk disimpan.

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

1. Menentukan akar penyebab kegagalan untuk setiap masalah bisnis.
2. Menghasilkan informasi mengenai titik penting penjualan berdasarkan kebiasaan pelanggan dalam membeli.
3. Menghitung kembali seluruh risiko yang ada dalam waktu yang singkat.
4. Mendeteksi perilaku penipuan yang dapat mempengaruhi organisasi.

2.2 Algoritma Hierarchical Clustering

Hierarchical Clustering Algorithm (HCA) adalah metode analisis kelompok yang berusaha untuk membangun sebuah hirarki dengan mengelompokkan data. Dengan mengelompokkan data-data tersebut, data pada kelompok yang sama memiliki kemiripan yang tinggi dan data pada kelompok yang berbeda memiliki kemiripan yang rendah [2]. Dalam reduksi data, *cluster* yang merepresentasikan data-data pada *cluster* tersebut akan digunakan untuk mengganti data-data mentah [2]. Seberapa efektif cara ini tergantung dengan sifat data yang ditangani. Data-data yang bisa dikelompokkan ke dalam *cluster* yang berbeda akan sangat cocok dengan cara ini [2]. Pada dasarnya HCA dibagi menjadi dua jenis yaitu *agglomerative (bottom-up)* dan *devisive (top-down)* [2]. Pendekatan *agglomerative* berusaha membentuk sebuah hierarki dengan menggabungkan *cluster*. Setiap objek akan dimasukkan kepada *cluster* tersendiri. Sebaliknya, pendekatan *devisive* akan berusaha memecah *cluster* untuk membentuk sebuah hierarki. Setiap objek berada pada satu *cluster* pada awalnya dan akan dipecah kepada *cluster* yang berbeda.

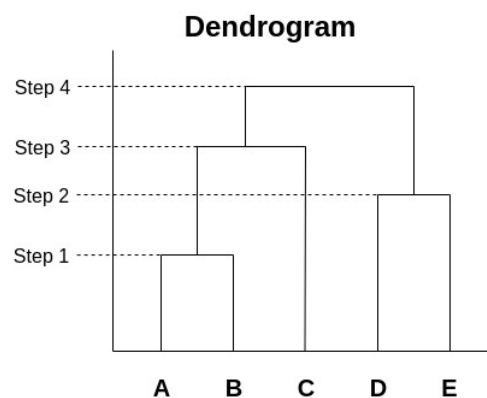
	A	B	C	D
A	0	2	5	6
B	2	0	7	3
C	5	7	0	4
D	6	3	4	0

Gambar 2.2: Matriks jarak

	(A,B)	C	D
(A,B)	0	?	?
C	?	0	4
D	?	4	0

Gambar 2.3: Matriks jarak

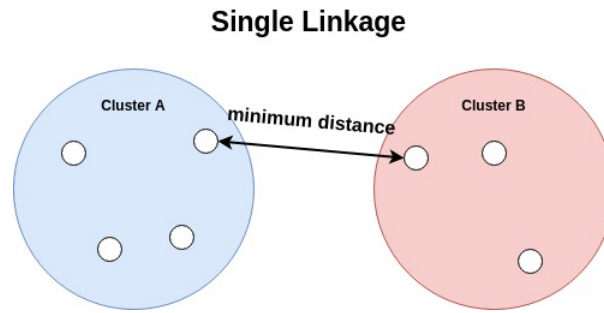
Pada *Hierarchical Agglomerative Clustering*, awalnya setiap objek akan dimasukkan kepada *cluster* tersendiri. Matriks jarak digunakan untuk merepresentasikan jarak antara *cluster*. Kemudian, dua buah *cluster* yang memiliki similaritas tertinggi akan digabungkan menjadi satu *cluster*. Similaritas antara *cluster* dapat dihitung dengan tiga metode yaitu *single linkage*, *complete linkage*, dan *centroid linkage* [3]. Pada Gambar 2.2, *cluster* A dan *cluster* B akan digabung menjadi satu karena jarak antara keduanya adalah terkecil dibanding dengan yang lainnya. Gambar 2.3 adalah hasil dari penggabungan *cluster* A dan *cluster* B dan matriks jarak harus dihitung kembali untuk mencari jarak baru antara *cluster* baru dengan *cluster* lainnya. Penggabungan *cluster* akan diulangi sampai tersisa satu *cluster*. *Hierarchical Agglomerative Clustering* akan hanya membutuhkan maksimal n iterasi. Hasil dari penggabungan *cluster* adalah sebuah hierarki. *Dendrogram* sangat umum digunakan untuk menggambarkan proses *Hierarchical Agglomerative Clustering*. Contoh *dendrogram* dapat dilihat pada Gambar 2.4.



Gambar 2.4: dendrogram

Berikut adalah penjelasan mengenai metode *single linkage*, *complete linkage*, dan *centroid linkage*:

- *Single linkage*: metode ini mencari jarak minimum dari perbandingan setiap anggota antara dua buah *cluster*. Bila ada dua buah *cluster* A dan *cluster* B, maka setiap anggota pada *cluster* A akan dihitung jaraknya kepada setiap anggota pada *cluster* B. Kemudian jarak minimum antara anggota akan diambil sebagai hasilnya. Untuk menghitung jarak antara anggota dapat digunakan *euclidean distance*, *manhattan distance*, atau ruang metrik lainnya. Ruang metrik yang digunakan disesuaikan dengan kebutuhan dan atribut dari data. Contoh *single linkage* dapat dilihat pada Gambar 2.5.

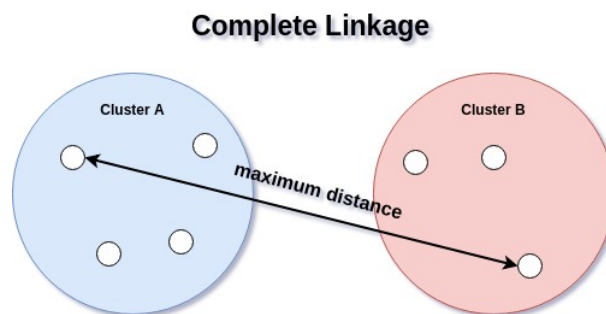
Gambar 2.5: Metode *single linkage*

Berikut adalah rumus untuk *single linkage*:

$$\min\{d(a,b) : a \in A, b \in B\},$$

dengan a dan b merupakan anggota dari *cluster* A dan B.

- *Complete linkage*: metode ini adalah kebalikan dari metode *single linkage*. Bila ada dua buah *cluster* A dan B, maka setiap anggota pada *cluster* A akan dihitung jaraknya kepada setiap anggota pada *cluster* B. Kemudian jarak maksimum antara anggota akan diambil sebagai hasilnya. Contoh *complete linkage* dapat dilihat pada Gambar 2.6.

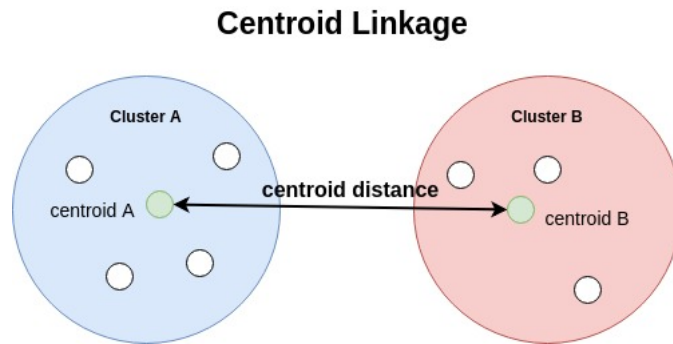
Gambar 2.6: Metode *complete linkage*

Berikut adalah rumus untuk *complete linkage*:

$$\max\{d(a,b) : a \in A, b \in B\},$$

dengan a dan b merupakan anggota dari *cluster* A dan B.

- *Centroid linkage*: metode ini menghitung jarak antara *centroid* dari dua buah *cluster*. *Centroid* sebuah *cluster* didapatkan dengan menghitung rata-rata dari setiap atribut dari anggota pada *cluster*. Contoh *centroid linkage* dapat dilihat pada Gambar 2.7.

Gambar 2.7: Metode *centroid linkage*

Berikut adalah rumus untuk *centroid linkage*:

$$\|c_a - c_b\|,$$

dengan c_a dan c_b merupakan *centroid* dari *cluster A* dan *B*.

Tabel 2.1: Tabel Data Koordinat

Cluster	x	y
A	2	2
B	2	3
C	4	6
D	8	10

Sebagai contoh, diberikan data yang memiliki atribut berupa koordinat x dan y . Data dapat dilihat pada Tabel 2.1. Data tersebut akan diolah dengan algoritma *Hierarchical Agglomerative Clustering* menggunakan metode *single linkage* dan *euclidean distance* untuk menghitung jaraknya. Berikut adalah langkah-langkah penyelesaiannya.

- Pertama, hitung matriks jarak antara *cluster*. Karena setiap *cluster* hanya memiliki satu anggota pada awalnya, similaritas antara *cluster* dapat langsung dihitung menggunakan *euclidean distance*. Matriks jarak yang dihasilkan bisa dilihat pada Gambar 2.8.

	A	B	C	D
A	0	1.0	4.47	10.0
B	1.00	0	3.61	9.22
C	4.47	3.61	0	5.66
D	10.0	9.22	5.66	0

Gambar 2.8: Matriks jarak

Jarak antara *cluster* A dan *cluster* B dapat dihitung dengan cara seperti berikut:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad (2.1)$$

$$= \sqrt{(2 - 2)^2 + (2 - 3)^2} \quad (2.2)$$

$$= \sqrt{0 + 1} \quad (2.3)$$

$$= \sqrt{1} \quad (2.4)$$

$$= 1 \quad (2.5)$$

2. Selanjutnya, gabungkan dua *cluster* yang memiliki similaritas tertinggi. Pada contoh ini, *cluster* A yang dibandingkan terhadap *cluster* B memiliki nilai terkecil yaitu 1. Similaritas antara kedua *cluster* adalah yang tertinggi. Hasil dari penggabungan kedua *cluster* dapat dilihat pada Gambar 2.9.

	(A, B)	C	D
(A, B)	0	?	?
C	?	0	5.66
D	?	5.66	0

Gambar 2.9: Hasil penggabungan *cluster*

3. Setelah itu, matriks jarak harus dikalkulasi kembali untuk mencari similaritas antara *cluster* barunya yaitu (A,B) dengan yang lainnya. Untuk mengkalkulasi ulang antara *cluster* baru dengan *cluster* lainnya, digunakan metode *single linkage*. Pada tahap ini setiap anggota dari *cluster* (A,B) akan dihitung jaraknya terhadap *cluster* C dan *cluster* D. Nilai minimum akan diambil sebagai hasil perbandingannya karena metode yang digunakan adalah *single linkage*. Berikut adalah contoh perhitungan antara *cluster* (A,B) dengan *cluster* C menggunakan metode *single linkage*.

$$d(A,C) = \sqrt{(2 - 2)^2 + (4 - 6)^2} \quad (2.6)$$

$$= 4.47 \quad (2.7)$$

$$d(B,C) = \sqrt{(2 - 3)^2 + (4 - 6)^2} \quad (2.8)$$

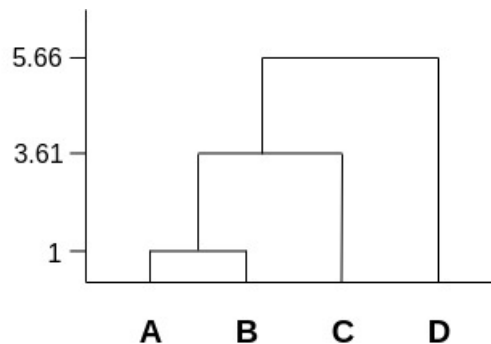
$$= 3.61 \quad (2.9)$$

- 1 Karena nilai 3.61 lebih kecil dari 4.47, maka nilai 3.61 diambil sebagai hasil. Contoh hasil dapat dilihat
 2 pada Gambar 2.10.

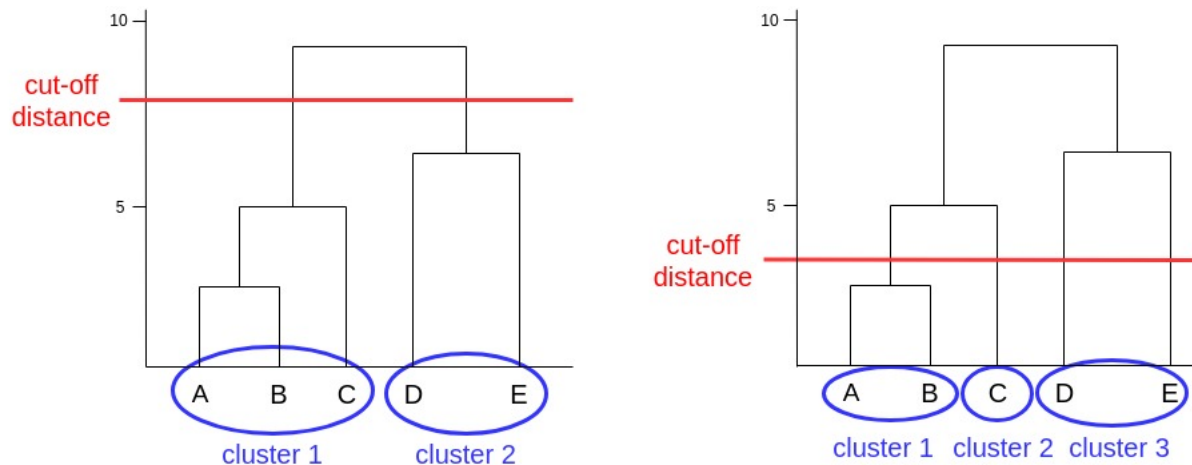
	(A, B)	C	D
(A, B)	0	3.61	?
C	3.61	0	5.66
D	?	5.66	0

Gambar 2.10: Hasil rekalkulasi

- 3 4. Ulangi langkah 2 dan 3 sampai satu *cluster* yang tersisa. Hasil akhir dalam bentuk *dendrogram* dapat
 4 dilihat pada Gambar 2.11.

Gambar 2.11: Hasil akhir *dendrogram*

- 5 Setelah *dendrogram* terbentuk, *dendrogram* perlu dipotong berdasarkan nilai *cut-off distance* yang diten-
 6 tukan. Nilai *cut-off distance* menentukan banyaknya *cluster* yang dihasilkan ketika memotong *dendrogram*.
 7 Semakin tinggi nilai *cut-off distance*, semakin dikit *cluster* yang dihasilkan dan sebaliknya. Berdasarkan
 8 Gambar 2.11, dapat dilihat bahwa nilai *cut-off distance* yang lebih tinggi menghasilkan *cluster-cluster* yang
 9 lebih sedikit. Sedangkan, nilai *cut-off distance* yang lebih rendah menghasilkan *cluster-cluster* yang lebih
 10 banyak. Perpotongan akan berdampak kepada hasil akhir ukuran data yang dihasilkan.

Gambar 2.12: Perpotongan *dendrogram*

- 1 Dari setiap *cluster* yang dihasilkan dari perpotongan, perlu dicari jumlah anggota pada *cluster*, nilai
- 2 minimum, maksimum, rata-rata, dan standar deviasi dari setiap atribut. Nilai-nilai ini dapat disebut pola.
- 3 Pola ini akan merepresentasikan dan menggambarkan karakteristik *cluster* tersebut. Pola ini akan disimpan
- 4 sebagai hasil akhir untuk menggantikan data mentahnya. Sebagai contoh diberikan sebuah *cluster* A yang
- 5 memiliki 4 anggota pada clusternya, setiap anggota memiliki 2 nilai atribut yang berbeda. Data untuk *cluster*
- 6 A dapat dilihat pada Tabel 2.2. Data pada tabel ini akan digunakan untuk mencari pola untuk *cluster* A.

Tabel 2.2: Tabel Contoh Data Cluster

Cluster A		
No	atribut 1	atribut 2
1	2	1
2	4	5
3	5	10
4	6	7

- 7 Hasil pola dari *cluster* A dapat dilihat pada Tabel 2.3.

Tabel 2.3: Tabel Hasil Pola Cluster A

jumlah anggota pada cluster	4	
	atribut 1	atribut 2
nilai minimum	2	1
nilai maksimum	6	10
nilai rata-rata	4.25	5.75
standar deviasi	1.479	3.269

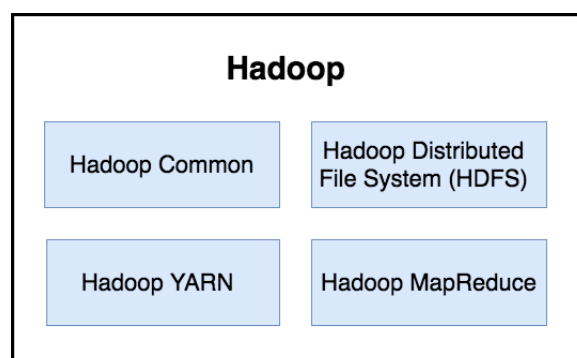
8 2.3 Hadoop

9 2.3.1 Penjelasan Hadoop

- 10 Hadoop dikembangkan oleh Doug Cutting dan Mike Cafarella pada tahun 2005 yang saat itu bekerja di
- 11 Yahoo. Nama Hadoop diberikan berdasarkan mainan 'Gajah' anak dari Doug Cutting. Hadoop adalah sebuah
- 12 *framework* atau platform *open source* berbasis Java. Hadoop memiliki kemampuan untuk penyimpanan

dan memproses data dengan skala yang besar secara terdistribusi pada *cluster*. *Cluster* tersebut terdiri dari perangkat keras komoditas [4]. Hadoop menggunakan teknologi Google MapReduce dan Google File System (GFS) sebagai fondasinya [5]. Beberapa karakteristik yang dimiliki Hadoop adalah sebagai berikut:

1. *Open Source*: Hadoop merupakan proyek *open source* dan kodenya bisa dimodifikasi sesuai kebutuhan.
2. *Distributed computing*: Data disimpan secara terdistribusi pada *Hadoop Distributed File System* (HDFS) dan data diproses secara paralel pada *node-node* di *cluster*.
3. *Fast*: Hadoop sangat cocok untuk melakukan *batch processing* bervolume besar karena mampu melakukannya secara paralel.
4. *Fault Tolerance*: Hadoop melakukan duplikasi data di beberapa *node* yang berbeda. Ketika sebuah *node* gagal memproses data, *node* yang memiliki duplikat data dapat menggantikannya untuk memproses data tersebut.
5. *Reliability*: Kegagalan mesin bukan masalah bagi Hadoop karena adanya duplikasi data.
6. *High availability*: Data dapat diambil dari sumber yang lain meskipun kegagalan mesin karena adanya duplikasi data.
7. *Scalability*: Hadoop dapat menambahkan *node* yang lebih banyak ke dalam *cluster* dengan mudah.
8. *Flexibility*: Hadoop dapat menangani data terstruktur maupun data tidak terstruktur.
9. *Economic and cost effective*: Hadoop tidak mahal karena berjalan pada *cluster* yang terdiri dari perangkat keras komoditas.
10. *Easy to use*: Hadoop mempermudah pengguna dalam merancang program paralel. Hadoop sudah menangani hal-hal terkait komputasi terdistribusi.
11. *Data locality*: Algoritma MapReduce akan didekatkan kepada *cluster* dan tidak sebaliknya. Ukuran data yang besar lebih sulit untuk dipindahkan dibanding ukuran algoritma yang kecil.



Gambar 2.13: Modul-modul Hadoop

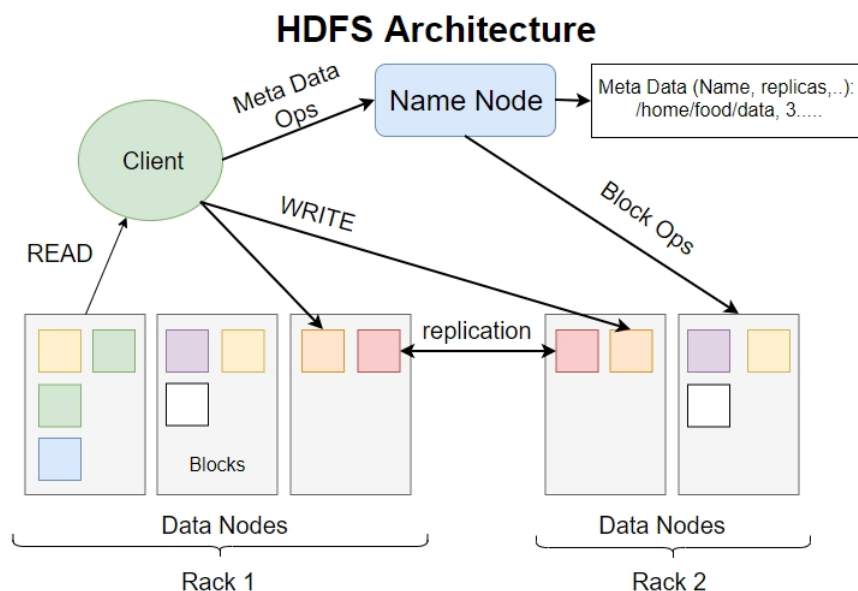
Berdasarkan gambar diatas (Gambar 2.13), *framework* Apache Hadoop terdiri dari beberapa modul . Modul-modul tersebut membentuk dan membantu untuk memproses data yang berskala besar. Modul-modul tersebut diantaranya adalah [5]:

1. *Hadoop Common*, module ini terdiri dari *library* dan *tools* yang dibutuhkan module Hadoop lainnya.
2. *Hadoop Distributed File System (HDFS)*, sebuah sistem-file terdistribusi milik Hadoop untuk penyimpanan data.
3. *Hadoop YARN*, *resource-management platform* yang bertanggung jawab untuk mengatur sumber daya pada *cluster*.
4. MapReduce, sebuah model pemrograman untuk pemrosesan skala besar.

2.3.2 Hadoop Distributed File System (HDFS)

Hadoop Distributed File System (HDFS) adalah sistem file terdistribusi yang dirancang untuk berjalan pada perangkat keras komoditas [5]. HDFS berbeda dari sistem file terdistribusi lainnya karena sifat *fault tolerance* 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 *data set* 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 data dari file sistem. 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 2.14: Arsitektur HDFS

Hadoop mengimplementasikan arsitektur *Master Slave* pada komponen primernya yaitu HDFS dan MapReduce [5]. Berdasarkan (Gambar 2.14), *master node* atau disebut NameNode bertugas untuk mengatur operasi-operasi seperti membuka, menutup, dan menamakan kembali file atau direktori pada sistem file.

Selain itu, NameNode meregulasi akses pengguna terhadap file dan mengatur block mana yang akan diolah oleh DataNode [5]. NameNode membuat semua keputusan terkait replikasi blok. NameNode secara berkala menerima *heart beat* dan *block report* dari masing-masing DataNode di *cluster*. *Heartbeat* mengimplikasikan bahwa DataNode berfungsi dengan benar.

Slave node atau dapat disebut DataNode merupakan pekerja dari HDFS [5]. DataNode bertanggungjawab untuk menjalankan perintah membaca dan menulis untuk sistem file Hadoop. NameNode bisa membuat, menghapus, dan mereplikasi block ketika diberi instruksi dari *master node*. DataNode menyimpan dan mengambil blok ketika diperintahkan oleh NameNode. Selain itu, DataNode melaporkan daftar blok-blok yang disimpan kepada NameNode secara rutin.

HDFS dirancang untuk menyimpan file yang berukuran sangat besar di seluruh mesin dalam *cluster* yang besar [5]. HDFS menyimpan setiap file sebagai blok yang berurutan. semua blok dalam file kecuali blok terakhir memiliki ukuran yang sama. Bisa dilihat pada (Gambar 2.14) 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.

Berikut adalah perintah-perintah dasar yang dapat digunakan untuk HDFS [6]:

- Command untuk membuat direktori HDFS untuk penyimpanan file.

```
$ hadoop fs -mkdir <dir-path>
```

- Command untuk melihat daftar konten direktori dari path yang diberikan.

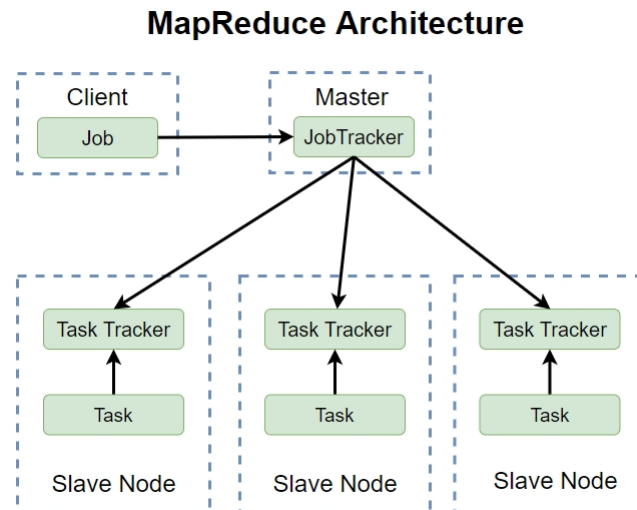
```
$ hadoop fs -ls
```

- Command untuk memasukan file atau direktori lokal kepada file sistem destinasi di dalam HDFS.

```
$ hadoop fs -put <localSrc> <dest>
```

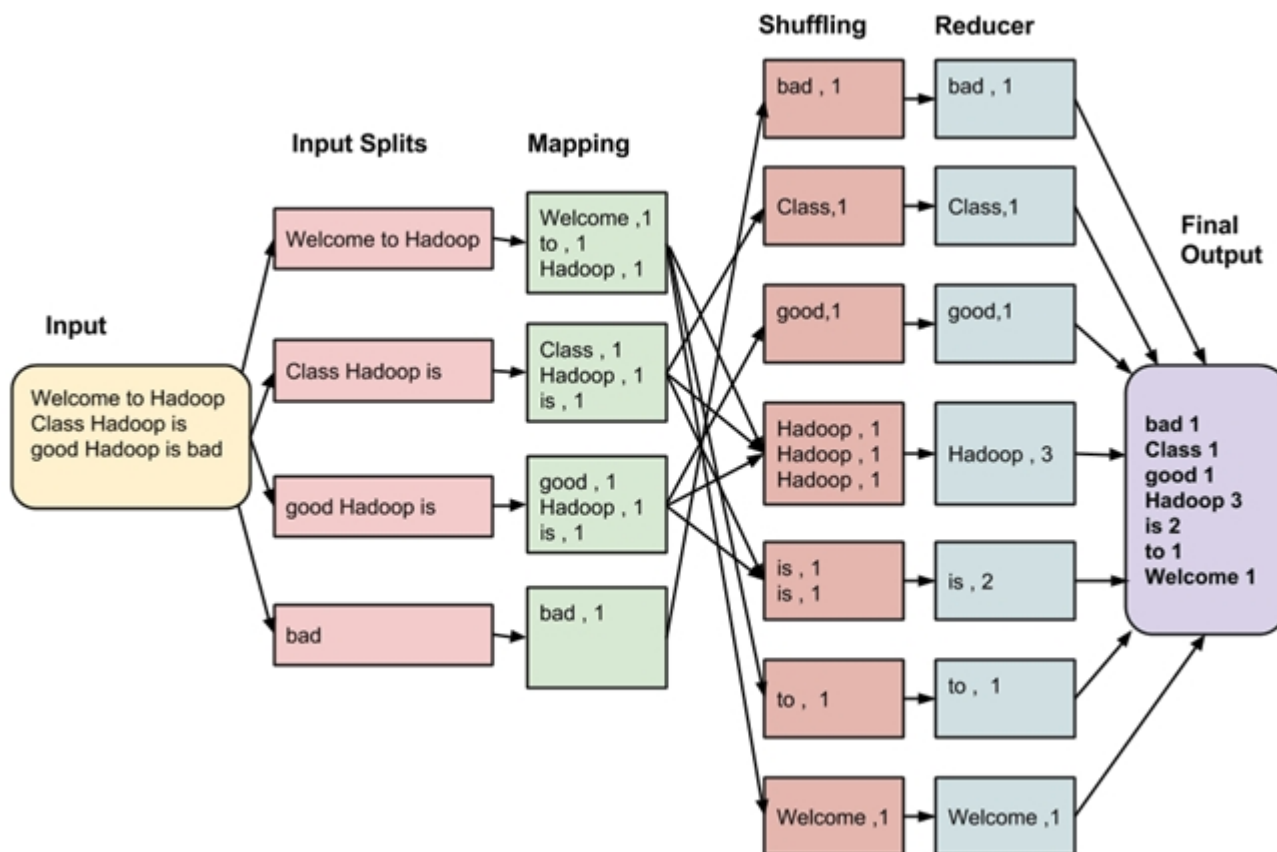
2.3.3 MapReduce

MapReduce adalah sebuah model pemrograman untuk memproses data berukuran besar secara terdistribusi dan paralel dalam *cluster* yang terdiri atas banyak komputer. Dalam memproses data, secara garis besar MapReduce dapat dibagi dalam dua proses yaitu proses *map* dan proses *reduce* [5]. Setiap fase memiliki pasangan *key-value* sebagai *input* dan *output* [5]. Kedua jenis proses ini didistribusikan atau dibagi-bagikan ke setiap komputer dalam suatu cluster dan berjalan secara paralel tanpa saling bergantung satu sama 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 2.15: Arsitektur MapReduce

- 1 Dapat dilihat pada (Gambar 2.15) yaitu arsitektur MapReduce. Pada arsitektur ini, *master node* disebut
- 2 JobTracker dan *slave node* disebut TaskTracker. JobTracker adalah jembatan antara pengguna dan fungsi
- 3 *map* maupun *reduce*. Ketika sebuah pekerjaan *map* atau *reduce* diterima oleh JobTracker, pekerjaan tersebut
- 4 akan dimasukkan ke dalam antrian. Pekerjaan dalam antrian akan dikerjakan sesuai urutan masuk pekerjaan
- 5 tersebut. Kemudian, pekerjaan akan ditugaskan kepada TaskTracker oleh JobTracker [5]. TaskTracker akan
- 6 mengeksekusi pekerjaan yang diberikan oleh JobTracker dan mengembalikan laporan kemajuan kepada
- 7 JobTracker [5].



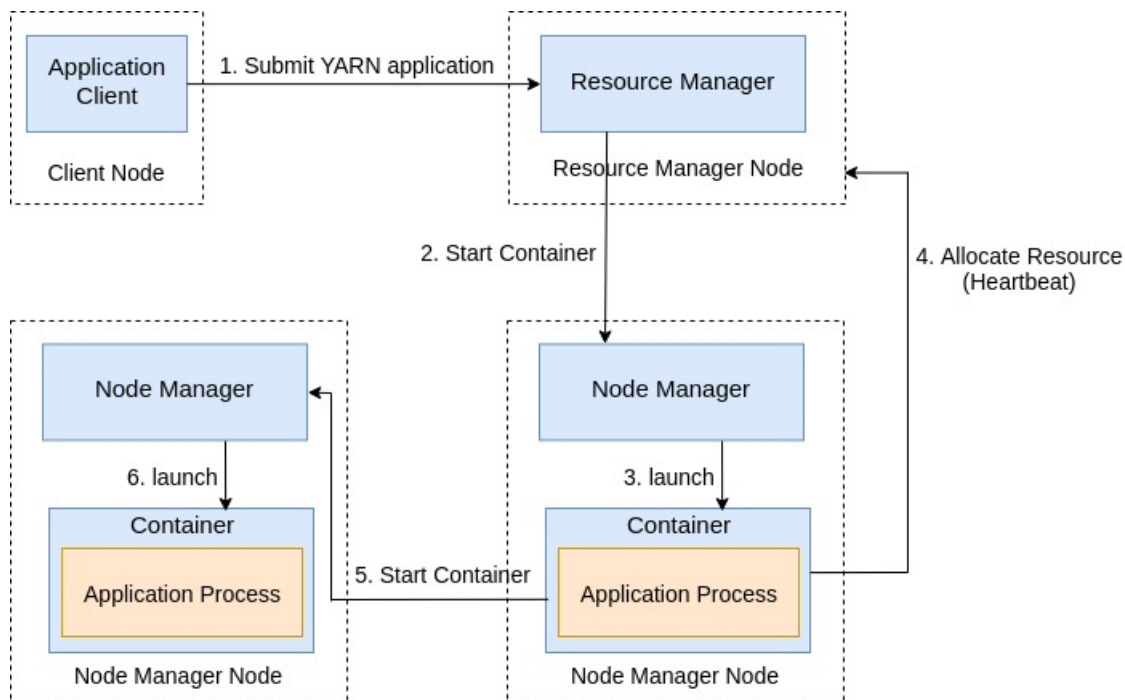
Gambar 2.16: Proses MapReduce

Berdasarkan (Gambar 2.16), berikut adalah langkah-langkah proses awal sampai akhir dari MapReduce:

1. *Input* dibagi menjadi *input split* yang berukuran sama. Setiap *input splits* akan dibuatkan *map task*.
2. Pada fase *map*, data pada setiap *split* akan dihitung berapa banyak kemunculan kata tersebut dan dijadikan pasangan $\langle \text{word}, \text{frequency} \rangle$ sebagai *output*.
3. Setelah fase *mapping* adalah fase *shuffling*. Tahap ini akan mengirim *output* dari fase *map* kepada *reducer*. Hasil dari fase *map* akan dikelompokkan berdasarkan *key* dan dibagi di antara *reducer*. Dalam contoh ini, kata-kata yang sama disatukan bersama dengan frekuensi masing-masing.
4. Terakhir adalah fase *reduce* dimana *output* dari *shuffling* akan dikumpulkan. Nilai-nilai dari fase *shuffling* akan digabungkan menjadi sebuah *output*. *Output* akan disimpan pada HDFS.

2.3.4 YARN

Apache YARN (Yet Another Resource Negotiator) adalah pengatur sumber daya dari *cluster* Hadoop. YARN bertujuan untuk memisahkan fungsionalitas antara pengaturan sumber daya dan penjadwalan pekerjaan. YARN memiliki dua tipe *daemon* yaitu *Resource Manager* dan *Node Manager* [5]. *Resource Manager* bertugas untuk mengatur sumber daya di seluruh *cluster* dan *Node Manager* yang berjalan pada *node*. *Node Manager* bertugas untuk menjalankan dan memantau *container* [5]. *Container* bertugas untuk mengeksekusi proses aplikasi yang spesifik.



Gambar 2.17: Proses menjalankan aplikasi pada YARN

Berikut adalah Gambar 2.17 yang mengabarkan langkah-langkah proses ketika menjalankan aplikasi pada YARN. Untuk menjalankan aplikasi pada YARN, *client* akan meminta *Resource Manager* untuk menjalankan proses aplikasi *master* (langkah 1). Kemudian, *Resource Manager* akan mencari *Node Manager* yang bisa menjalankan aplikasi *master* dalam sebuah *container* (langkah 2 dan 3). Ketika aplikasi *master* sudah berjalan, aplikasi *master* bisa melakukan komputasi pada *container* dan mengembalikan hasil kepada *client*. Selain itu, aplikasi *master* bisa saja meminta sumber daya tambahan (langkah 4) dan menggunakan sumber daya tersebut untuk komputasi terdistribusi (langkah 5 dan 6).

2.4 Spark

2.4.1 Pembahasan Umum Spark

Apache spark adalah sebuah *cluster computing platform* dirancang untuk kecepatan dan *general-purpose* [7]. Spark dirancang berdasarkan model MapReduce yang populer untuk memberikan dukungan yang efisien kepada banyak tipe komputasi, termasuk *interactive query*, dan *stream processing* [7]. 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 kemampuannya untuk melakukan *in memory computations* [7]. Selain itu, sistem Spark lebih efisien daripada MapReduce dalam menjalankan aplikasi yang rumit pada disk.

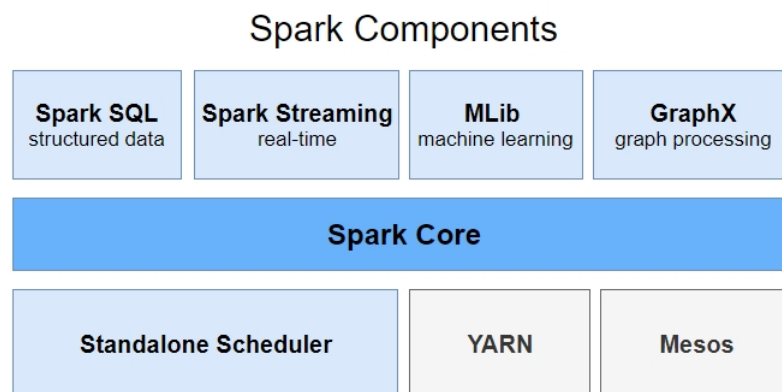
Pada sisi *general-purpose*, Spark dirancang untuk mencakup berbagai beban kerja yang sebelumnya diperlukan sistem terdistribusi terpisah, termasuk aplikasi *batch*, *iterative algorithms*, *interactive query*, dan *streaming*. Dengan mendukung beban kerja tersebut di mesin yang sama, Spark membuat pekerjaan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

beban dalam merawat *tools* yang terpisah.

Spark dirancang agar sangat mudah diakses dengan memberikan API sederhana untuk Python, Java, Scala, dan SQL [7]. Spark dengan mudah berintegrasi dengan *tools Big Data* lainnya, terutama Hadoop. Spark bisa berjalan pada Hadoop *cluster* dan mengakses sumber data Hadoop mana saja.

2.4.2 Komponen Spark

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 task-task komputasi tersebar di banyak pekerja, mesin, atau *cluster* [7]. Karena *core engine* dari Spark sangat cepat dan dirancang untuk tujuan umum, Spark menjalankan banyak komponen di level yang lebih tinggi untuk menangani berbagai macam pekerjaan khusus seperti SQL atau *machine learning* [7]. Komponen-komponen ini dirancang untuk saling beroperasi dengan erat, Spark membiarkan pengguna untuk menggabungkan komponen seperti *library* dalam suatu proyek perangkat lunak.



Gambar 2.18: Komponen pada Spark

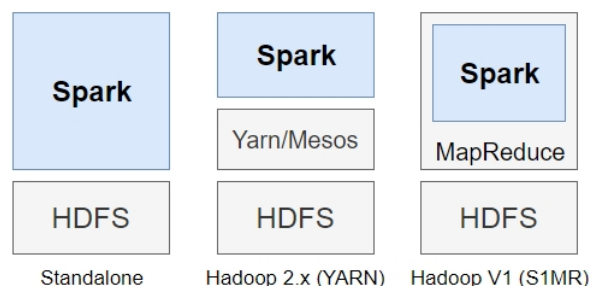
Berdasarkan (Gambar 2.18), Spark memiliki beberapa komponen sebagai berikut:

- **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 [7]. Spark Core memiliki banyak API *resilient distributed datasets*(RDD), yang merupakan abstraksi pemrograman utama Spark [7]. RDD mewakili suatu koleksi *item* yang didistribusikan di banyak node komputasi yang dapat dimanipulasi secara paralel [7]. Spark Core menyediakan banyak API untuk membangun dan memanipulasi ini koleksi.
- **Spark SQL:** Spark SQL adalah sebuah modul untuk bekerja dengan data yang terstruktur module [7]. Modul ini memungkinkan melakukan kueri pada data terstruktur melalui SQL serta varian Apache Hive dari SQL disebut Hive Query Language (HQL) dan mendukung banyak sumber data, termasuk tabel Hive, Parquet, 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* [7]. Contoh *data stream* 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 Core's RDD API untuk memanipulasi aliran data. Hal ini membuat *developer* mudah mempelajari 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* [7]. 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* [7]. Seperti Spark Streaming dan Spark SQL, GraphX memperluas API Spark RDD, memungkinkan kita untuk membuat *directed graph* dengan *arbitrary properties* yang melekat pada setiap *vertex* dan *edge*. GraphX juga 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 dapat ditambah secara efisien dari satu hingga ribuan node komputasi. Untuk mencapai hal ini dan memaksimalkan fleksibilitas, Spark dapat menjalankan lebih dari satu variasi manajer *cluster* seperti Hadoop YARN, Apache Mesos, *simple cluster manager* pada diri Spark sendiri yang disebut *Standalone Scheduler*.

2.4.3 Tiga Cara Membangun Spark di Atas Hadoop



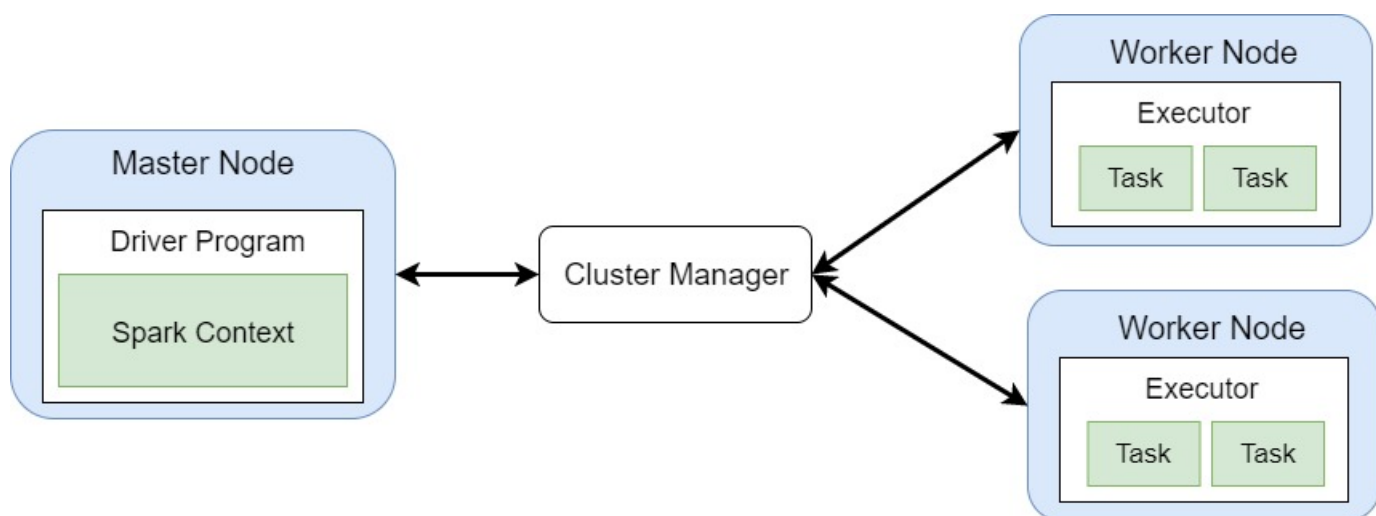
Gambar 2.19: Macam-macam cara instalasi Spark

1 Ada tiga cara untuk menginstal Spark berdasarkan Gambar 2.19 diatas, ketiga cara tersebut akan dijelaskan
2 dibawah:

- 3 • *Standalone*: Spark *standalone* berarti Spark menempati tempat di atas HDFS (Hadoop Distributed File
4 System) dan ruang dialokasikan untuk HDFS, secara eksplisit. Spark dan MapReduce akan berjalan
5 berdampingan untuk mencakup semua pekerjaan percikan di cluster [7].
- 6 • Hadoop Yarn: Spark berjalan pada Yarn tanpa perlu pra-instalasi atau akses root [7]. Cara ini membantu
7 mengintegrasikan Spark ke dalam ekosistem Hadoop atau Hadoop stack. Cara ini memungkinkan
8 komponen lain untuk berjalan di atas tumpukan.
- 9 • Spark pada MapReduce: Spark pada MapReduce digunakan untuk menjalankan job-job pada spark
10 selain untuk *standalone deployment* [7]. Dengan adanya SIMR, pengguna dapat memulai Spark dan
11 menggunakan *Spark Shell* tanpa akses administratif.

13 2.4.4 Arsitektur Spark

14 Pada bagian ini akan dijelaskan arsitektur Spark. Spark menggunakan arsitektur *master* dan *slave*. Sebuah
15 Spark *cluster* memiliki satu *master* dan banyak *slave* atau bisa disebut sebagai *worker*. Spark memiliki
16 beberapa komponen peting dalam arsitekturnya seperti *Driver Program*, *Spark Context*, *Cluster Manager*, .
17 Gambar 2.20 menggambarkan komponen-komponen arsitektur Spark.



Gambar 2.20: Arsitektur Spark

18 Berikut adalah penjelasn dari komponen-komponen Gambar Gambar 2.20:

- 19 • **Driver Program**
20 *Driver program* yang berjalan pada *master node* bertugas menjalankan fungsi `main()` dari aplikasi dan
21 tempat dimana *Spark Context* dibuat. Kode program akan diterjemahkan menjadi *tasks* dan dijadwalkan
22 kepada *executors* untuk dikerjakan. *Driver program* akan berkomunikasi dengan
- 23 • **Cluster Manager** untuk mengatur sumber daya pada *cluster*.

- **Spark Context**

Spark Context menghubungkan pengguna dengan *cluster*. *Spark Context* dapat terhubung dengan beberapa cluster manager seperti YARN, MESOS, dan *Spark standalone cluster manager*. *Spark Context* dapat digunakan untuk membuat *Resilient Distributed Datasets* (RDD), *accumulators*, dan *broadcast variable*.

- **Cluster Manager**

Cluster Manager berfungsi mengatur sumber daya pada sebuah *cluster*. Spark dapat berjalan pada berbagai macam *cluster manager* seperti Apache Mesos, Hadoop YARN, dan *Spark's stand alone*. *Cluster manager* akan berusaha mendapatkan sumber daya pada *cluster* dan mengalokasikannya kepada *Spark job* yang sedang berjalan.

- **Executors**

Executors adalah proses-proses yang berjalan pada *worker node* bertanggung jawab untuk mengerjakan *tasks* yang diberikan. *Executors* dibuat ketika aplikasi dijalankan dan akan tetap ada selama aplikasi itu berjalan.

- **Tasks**

Task adalah sebuah satuan kerja pada Spark. *Task* berisi perintah. Perintah tersebut merupakan fungsi yang diserialisasi. *Task* akan dikirimkan oleh *driver program* kepada *executor*. Kemudian, *executor* akan mendeserialisasi perintah tersebut dan mengerjakannya. Biasanya *task* akan dibuat untuk setiap partisi. Partisi merupakan potongan data yang terdistribusi pada *cluster*.

<https://www.dezyre.com/article/apache-spark-architecture-explained-in-detail/338>

2.4.5 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* [7]. 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. [7]

Secara umum, RDD adalah kumpulan *read-only, partitioned collection* dari *records*. RDD dapat dibuat melalui operasi deterministik dari data pada penyimpanan yang stabil atau RDD lainnya [7]. 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* kepada HDFS.

Untuk menangani masalah tersebut, dibangun *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 di antara *jobs*. *Data sharing* dalam memori lebih cepat 10 hingga 100 kali lipat dibanding *network* atau *disk*.

Berikut adalah sifat-sifat dari RDD :

- *In Memory*: Data pada RDD disimpan pada memori sebesar mungkin dan selama mungkin.
- *Partitioned*: *records* dipartisi dan didistribusikan kepada *node-node* di dalam *cluster*.
- *Typed*: RDD memiliki tipe data seperti RDD[Long], RDD[String] dan tipe data lainnya.
- *Lazy evaluation*: Data didalam RDD tidak akan tersedia atau berubah sampai sebuah perintah *action* telah dieksekusi.
- *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 paralel.
- *Cacheable*: Pengguna dapat memilih RDD mana yang akan dipakai kembali dan memilih tempat penyimpanan yaitu memori atau disk. Dengan begitu, data dapat diakses lebih cepat untuk permintaan selanjutnya.

Ada dua cara untuk membuat sebuah RDD. Cara pertama adalah dengan memuat dataset eksternal [7]. Sedangkan cara alternatif adalah dengan mendistribusikan sebuah koleksi objek seperti *list* atau *set* [7]. Ketika sebuah RDD telah dibuat, ada dua tipe operasi yaitu *transformations* dan *actions* yang bisa dilakukan RDD. *Transformations* membuat RDD baru dari RDD sebelumnya [7]. Berbeda dengan *transformations*, *actions* mengembalikan nilai hasil komputasi berdasarkan RDD [7]. Hasil dari *actions* akan dikembalikan kepada *driver program* atau disimpan pada penyimpanan eksternal seperti HDFS.

Berikut adalah contoh pembuatan RDD dari sumber eksternal dan koleksi objek:

- Contoh pembuatan RDD dari sumber eksternal.

```
val lines = sc.textFile("/path/to/README.md")
```
- Contoh pembuatan RDD dari koleksi objek.

```
val lines = sc.parallelize(["a", "b", "c", "d", "e"])\
```

Transformations pada RDD adalah sebuah operasi yang menerima RDD sebagai 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, Spark akan mencatat *transformation* apa saja yang dilakukan pada RDD awal. *Transformations* akan dieksekusi ketika sebuah *actions* dipanggil.

Berikut adalah contoh *filter transformation* di Scala. *Filter* digunakan untuk menyaring elemen-elemen yang sesuai dengan kriteria yang ditentukan. Pada kasus ini, filter akan mengambilkan baris-baris yang memiliki kata *error*.

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

Berikut adalah Tabel 2.4 berisi daftar *transformations* yang umum pada Spark:

Tabel 2.4: Tabel transformations

<i>Transformations</i>	Penjelasan
map (func)	Mengembalikan RDD baru yang dibentuk dengan melewati setiap elemen melalui fungsi func.
mapPartitions (func)	Mengembalikan RDD baru yang dibentuk dengan melewati setiap partisi melalui fungsi func.
filter (func)	Mengembalikan RDD baru yang dibentuk dengan memilih elemen-elemen yang mengembalikan nilai <i>true</i> dari fungsi func.
flatMap (func)	Mirip dengan <i>map</i> , tetapi setiap elemen dapat dipetakan menjadi nol atau lebih elemen sebagai keluaran.
union (otherDataset)	Mengembalikan RDD baru yang mengandung elemen dari kedua sumber.
intersection (otherDataset)	Mengembalikan RDD baru yang berisi potongan elemen dari sumber dan sumber lainnya.
distinct ([numPartitions])	Mengembalikan RDD baru yang mengandung elemen yang unik dari sumber.
groupByKey ([numPartitions])	Mengembalikan RDD baru bertipe <i>pairs</i> (K, Iterable<V>) dari sumber RDD bertipe (K, V).
groupByKey (func,[numPartitions])	Mengembalikan RDD baru berupa <i>pairs</i> (K, V) yang sudah diagregasi berdasarkan <i>key</i> dan fungsi <i>reduce</i> yang diberikan.
sortByKey ([ascending], [numPartitions])	Mengembalikan RDD baru berupa <i>pairs</i> (K, V) yang terurut secara menaik atau menurun berdasarkan parameter boolean yang diberikan.
join (otherDataset, [numPartitions])	Mengembalikan gabungan RDD berupa <i>pairs</i> (K, V) dan (K, W) menjadi <i>pairs</i> (K, (V,W)).
cogroup (otherDataset, [numPartitions])	Mengembalikan RDD berupa <i>tuples</i> (K, (Iterable<V>, Iterable<W>)) dari <i>pairs</i> (K, V) dan (K, W).
cartesian (otherDataset)	Mengembalikan RDD berupa <i>pairs</i> (T, U) dari <i>dataset</i> T dan U.

Berikut adalah contoh operasi *action* pada RDD. Pada contoh ini, fungsi *reduceByKey* digunakan untuk menghitung jumlah kata yang ada.

```

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

```

Actions merupakan operasi yang mengembalikan sebuah nilai kepada *driver program* atau tempat penyimpanan eksternal. Untuk mengembalikan sebuah nilai, kita bisa menggunakan *take()*, *count()*, *collect()*, dan *actions* lainnya. Operasi *take()* digunakan untuk mengambil sebagian kecil elemen pada RDD. Ketika menggunakan *collect()*, memori pada satu komputer harus cukup untuk menampung seluruh *data set* [7]. Operasi tersebut sebaiknya digunakan pada *data set* yang berukuran kecil. *Data set* yang berukuran besar dapat disimpan pada tempat penyimpanan eksternal. Setiap kali sebuah *actions* dipanggil, seluruh RDD akan dikomputasi dari akarnya. Untuk mencapai efisiensi yang lebih tinggi, bisa dilakukan *persist* terhadap

intermediate results.

Berikut adalah Tabel 2.5 berisi daftar *actions* yang umum pada Spark:

Tabel 2.5: Tabel Actions

<i>Actions</i>	Penjelasan
reduce(func)	Mengagregasikan seluruh elemen pada RDD menggunakan fungsi yang diberikan pada <i>parameter</i> .
collect()	Mengembalikan seluruh <i>data set</i> sebagai <i>array</i> kepada <i>driver program</i> .
count()	Mengembalikan jumlah elemen pada RDD.
first()	Mengembalikan elemen pertama pada RDD.
take(n)	Mengembalikan sebuah <i>array</i> dengan n jumlah elemen pertama dari RDD.
takeOrdered(n, [ordering])	Mengembalikan sebuah <i>array</i> dengan n jumlah elemen pertama dari RDD secara terurut.
saveAsTextFile(path)	Menyimpan <i>dataset</i> sebagai <i>text file</i> pada direktori yang ditentukan.
saveAsSequenceFile(path)	Menyimpan RDD sebagai Hadoop SequenceFile pada direktori yang ditentukan.
saveAsObjectFile(path)	Menyimpan RDD sebagai format yang sederhana menggunakan Java Serialization pada direktori yang ditentukan.
countByKey()	Menjumlahkan <i>pairs</i> (K, V) berdasarkan <i>key</i> dan mengembalikan sebuah <i>pairs</i> berisi (K, int).
foreach(func)	Memproses setiap elemen pada RDD menggunakan fungsi <i>func</i> yang diberikan.

2.5 Scala

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 "Scalable Language". 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`.

2.5.1 Expressions

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

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

3 *Ekspressions* dapat dikembalikan dengan perintah `println`.

```
4 println(1)
5 println(100) // 100
6 println(1 + 1) // 2
7 println("Hi!") // Hi!
```

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

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

15 *variable* mirip dengan `value`, tetapi nilai *variable* dapat diisi kembali.

```
16 var x = 2 + 2
17 x = 4
18 println(x) // 4
19 x = 7
20 println(x) // 7
```

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

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

25 2.5.2 Blocks

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

```
27 println({
28     val x = 1 + 1
29     x + 1
30 }) // 3
```

31 2.5.3 Loop dan Conditional

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


```
1  for( var x <- Range ){
2      statement(s);
3  }
4
5  var x = 0
6  while (x < 10) {
7      println(x)
8      x += 1
9  }
```

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

```
13 if( x < 20 ){
14     println("This is if statement");
15 }
16
17 if( x < 20 ){
18     if( x < 5) {
19         println("smallest");
20     }
21 }
22
23 if( x < 10 ){
24     println("This is bigger");
25 } else {
26     println("This is smaller");
27 }
28
29 if( x == 1 ){
30     println("1");
31 } else if (x == 2){
32     println("2");
33 }
```

34 2.5.4 Functions

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

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

39

```
1 val addOne = (x: Int) => x + 1 // function biasa
2 println(addOne(2)) // 3
3
4 val add = (x: Int, y: Int) => x + y
5 println(add(1, 2)) // 3
```

6 Pada sisi sebelah kiri tanda `=>` adalah parameter-parameter sebuah *function*, pada sisi sebelah kanan
7 merupakan ekspresi-ekspresi yang melibatkan parameter tersebut.

9 2.5.5 Methods

10 *Method* sangat mirip dengan *function*, tetapi *method* memiliki beberapa perbedaan. *Method* harus di
11 definisikan dengan kata kunci `def`, diikuti dengan nama *method*, parameter-parameter dari *method* tersebut,
12 tipe kembalian *method*, dan isi dari *method* tersebut.

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

15 *Method* bisa mempunyai lebih dari satu parameter.

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

18 *Method* dapat tidak memiliki parameter.

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

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

```
22 def getSquareString(input: Double): String = {
23     val square = input * input
24     square.toString
25 }
```

26 Ekspresi terakhir dari *method* menjadi nilai yang akan dikembalikan. *Scala* mempunyai *keyword* `return`,
27 tetapi sangat jarang digunakan.

28 2.5.6 Class dan Object

29 *Class* pada *Scala* didefinisikan dengan kata kunci *class* diikuti dengan namanya dan terakhir adalah *constru-*
30 *ctor* parameter.

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

```
1    }
2  }
3
```

4 Berikut adalah cara mendeklarasi objek pada Scala

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

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

```
9 object IdFactory {
10     private var counter = 0
11 Main method
12     def create(): Int = {
13         counter += 1
14         counter
15     }
16 }
17
18 val newId: Int = IdFactory.create()
19 println(newId) // 1
20 val newerId: Int = IdFactory.create()
21 println(newerId) // 2
22
```

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

25

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

```
27 object Main {
28     def main(args: Array[String]): Unit = {
29         println("Hello, Scala developer!")
30     }
31 }
```

32 2.5.7 Higher Order Function

33 Pada bahasa Scala ada yang disebut sebagai *Higher Order Function*. *Higher Order Function* merupakan
34 sebuah fungsi yang menerima fungsi lainnya sebagai *parameter* dan mengembalikan sebuah fungsi sebagai
35 hasilnya. Berikut adalah contoh-contoh *Higher Order Function*:

```
36 val salaries = Seq(20000, 70000, 40000)
37 val doubleSalary = (x: Int) => x * 2
```

```
1 val newSalaries = salaries.map(doubleSalary) // List(40000, 140000, 80000)
2
```

3 Kita bisa mempersingkat kode dengan sebuah fungsi *anonymous* dan langsung dimasukan pada *parameter*

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

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

```
8 case class WeeklyWeatherForecast(temperatures: Seq[Double]) {
9
10     private def convertCtoF(temp: Double) = temp * 1.8 + 32
11
12     def forecastInFahrenheit: Seq[Double] = temperatures.map(convertCtoF) }
13 }
```

14 Salah satu alasan untuk menggunakan *higher order function* adalah untuk mengurangi kode yang
15 berlebihan. Katakanlah ada beberapa metode yang dapat menaikkan gaji seseorang dengan berbagai faktor.
16 Tanpa membuat *higher order function*, kode akan terlihat seperti berikut:

```
17 object SalaryRaiser {
18
19     def smallPromotion(salaries: List[Double]): List[Double] =
20         salaries.map(salary => salary * 1.1)
21
22     def greatPromotion(salaries: List[Double]): List[Double] =
23         salaries.map(salary => salary * math.log(salary))
24
25     def hugePromotion(salaries: List[Double]): List[Double] =
26         salaries.map(salary => salary * salary)
27 }
```

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

```
30 object SalaryRaiser {
31
32     private def promotion(salaries: List[Double], promoF: Double => Double): List[Double] =
33         salaries.map(promotionFunction)
34
35     def smallPromotion(salaries: List[Double]): List[Double] =
36         promotion(salaries, salary => salary * 1.1)
37 }
```

```
1  def bigPromotion(salaries: List[Double]): List[Double] =
2      promotion(salaries, salary => salary * math.log(salary))
3
4  def hugePromotion(salaries: List[Double]): List[Double] =
5      promotion(salaries, salary => salary * salary)
6  }
```


BAB 3

STUDI DAN EKSPLORASI APACHE SPARK

3.1 Instalasi Apache Spark

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:

- Pastikan bahwa Java telah diinstal 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)
```

- Pastikan bahwa Scala telah diinstal dengan versi minimal 2.11.x. Berikut adalah perintah untuk memastikan bahwa Scala telah terinstal dengan versi yang benar:

```
$ scala -version
```

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

Bila Java dan Scala belum terinstal pada komputer, berikut adalah langkah-langkah instalasi Java dan Scala untuk kebutuhan Spark:

- Berikut adalah perintah-perintah untuk menginstal Java menggunakan terminal pada sistem operasi Linux:

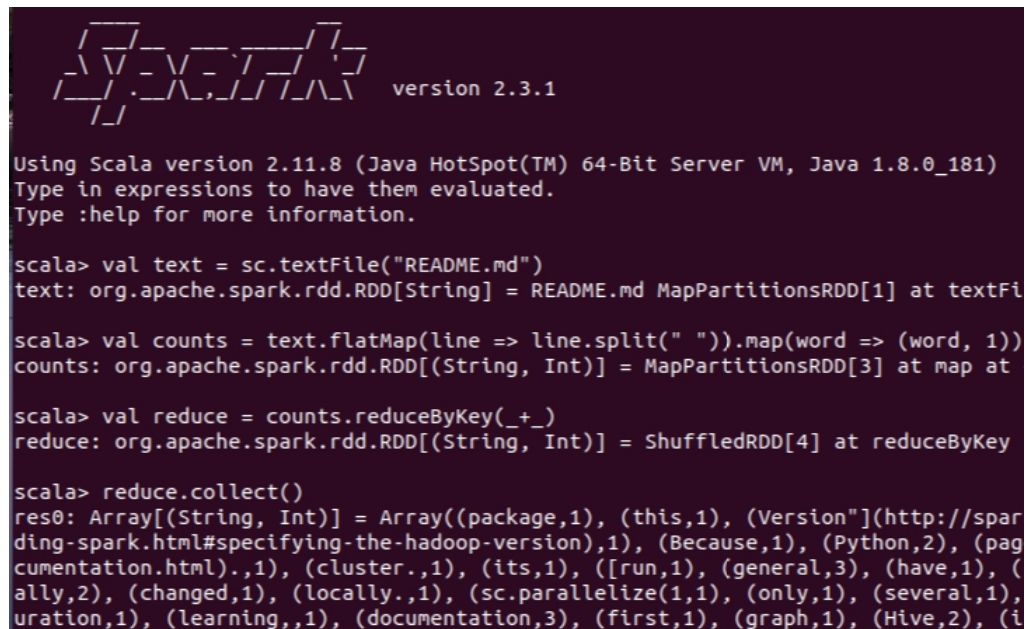
```
$ sudo apt-get update
```

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

- Berikut adalah perintah-perintah untuk menginstal Scala menggunakan terminal pada sistem operasi Linux:

3.2 Eksplorasi Spark Shell

- Pada bagian ini, penulis akan menjelaskan percobaan untuk menghitung jumlah setiap kata 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 *transformation* dan juga *action*.



```

version 2.3.1

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_181)
Type in expressions to have them evaluated.
Type :help for more information.

scala> val text = sc.textFile("README.md")
text: org.apache.spark.rdd.RDD[String] = README.md MapPartitionsRDD[1] at textFile at ...

scala> val counts = text.flatMap(line => line.split(" ")).map(word => (word, 1))
counts: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[3] at map at ...

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

scala> reduce.collect()
res0: Array[(String, Int)] = Array((package,1), (this,1), (Version,1)(http://spark.apache.org/docs/1.2.1/README.html#specifying-the-hadoop-version),1), (Because,1), (Python,2), (documentation,1), (cluster,1), (its,1), (run,1), (general,3), (have,1), (ally,2), (changed,1), (locally,1), (sc.parallelize(1,1), (only,1), (several,1), (uration,1), (learning,1), (documentation,3), (first,1), (graph,1), (Hive,2), (i

```

Gambar 3.2: Word Count

Berdasarkan gambar diatas (Gambar 3.2), berikut adalah langkah-langkah percobaan:

1. Pertama, jalankan spark shell dengan command berikut pada terminal:

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

2. Setelah itu, kita akan membuat text RDD dari sumber eksternal yaitu file README.md. Command dibawah digunakan untuk membuat RDD dari file eksternal:

```
scala> val text = sc.textFile("README.md")
```

Dapat dilihat bahwa RDD bertipe *String* telah sukses dibuat.

```
text: org.apache.spark.rdd.RDD[String] = README.md MapPartititonsRDD[1] at textFile at ...
```

3. Kemudian, kita akan memecahkan kalimat menjadi kata menggunakan operasi *transformation* `flatMap()`. Setelah itu, setiap kata akan dijadikan pasangan *key* (kata) dan *value* (kata,1). Berikut adalah perintah yang harus dijalankan:

```
val counts = text.flatMap(line => line.split(" ")).map(word => (word, 1))
counts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[3] ...
```

4. Langkah selanjutnya, kita akan menghitung jumlah setiap kata dengan cara berikut. Operasi `reduceByKey()` akan menjumlahkan kata dengan *key* yang sama. Contoh perintah dapat dilihat dibawah:

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

5. Terakhir, kita akan mengambil hasil dengan perintah operasi `collect()` yang merupakan sebuah *action*. Berikut adalah perintah yang harus dijalankan:

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

3.3 Instalasi Apache Spark pada multi-node cluster

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

1. Tambahkan entri dalam file host *master* dan *slave*. *Master* merupakan komputer utama dan *slaves* merupakan komputer pekerja. Berikut adalah perintah yang harus dijalankan:

```
$ sudo gedit /etc/hosts
```

Tambahkan IP *master* dan juga *slave* pada file.

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

2. Kemudian install Java pada setiap *master* dan *slave*, jangan lupa untuk memastikan versi java yang di install. Berikut adalah perintah untuk menginstal Java:

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

Pastikan versi Java yang di instal dengan perintah berikut:

```
$ java -version
```

3. Kemudian, instal Scala pada setiap master dan slave, jangan lupa untuk memastikan versi Scala yang diinstal.

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

Pastikan veri Scala yang diinstal dengan perintah berikut:

```
$ scala -version
```

4. Setelah melakukan instalasi Scala dan Java, maka kita perlu melakukan instalasi Open SSH Server-Client pada *master*. Berikut adalah perintah yang harus dijalankan:

```
$ sudo apt-get install openssh-server openssh-client
$ ssh-keygen -t rsa -P
```

5. Selanjutnya, kita perlu melakukan konfigurasi SSH pada *slave* dan juga *master*. Salin `.ssh/id_rsa.pub` milik *master* kepada `.ssh/authorized_keys` untuk *master* dan juga *slave*.

6. Setelah itu, kita akan mengunduh dan menginstal Spark pada setiap *slave* dan *master*. Berikut adalah langkah-langkah yang diikuti:

Unduh versi spark yang diinginkan pada <https://spark.apache.org/downloads.html>

Ekstrak spark dengan perintah berikut:

```
$ tar xvf spark-2.3.0-bin-hadoop2.7.tgz
$ sudo mv spark-2.3.0-bin-hadoop2.7 /home/user/spark
```

7. Setelah selesai menginstal Spark, kita harus mengubah file `.bashrc`.

Buka file `bashrc` dengan command berikut:

```
$ sudo gedit .bashrc
```

Tambahkan baris berikut pada file `.bashrc`:

```
export PATH = $PATH:/home/user/spark/bin
```

Jalankan perintah berikut untuk memastikan perubahan telah terjadi pada file `.bashrc`:

```
source .bashrc
```

8. Kemudian, Spark harus dikonfigurasi untuk *master* dengan mengubah file `spark-env.sh`. Berikut adalah perintah-perintah yang harus dijalankan

```
$ cd /home/user/spark/conf
$ cp spark-env.sh.template spark-env.sh
$ sudo gedit spark-env.sh
```

Tambahkan baris berikut pada file tersebut:

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

Kemudian edit file `slaves` pada `/home/user/spark/conf` dengan perintah berikut:

```
$ sudo gedit slaves
```

Tambahkan baris berikut pada file tersebut:

```
master
slave1
slave2
slave3
```

9. Sekarang, kita bisa menjalankan `spark cluster` dengan perintah berikut:

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

Untuk memberhentikananya masukan perintah berikut:

```
$ ./sbin/start-all.sh
```

3.4 Percobaan Spark Submit

Pada percobaan ini, kita akan mencoba mengumpulkan sebuah jar kepada `spark-submit`. Aplikasi yang dibuat harus memiliki konfigurasi Spark dan diubah menjadi jar untuk dikumpulkan kepada `spark-submit`. Aplikasi yang dibuat akan membaca file yang disediakan dan menghitung jumlah kata yang ada. Sebelum melakukan percobaan, ada beberapa kebutuhan yang harus dipenuhi. Berikut adalah kebutuhan-kebutuhan yang harus dipenuhi:

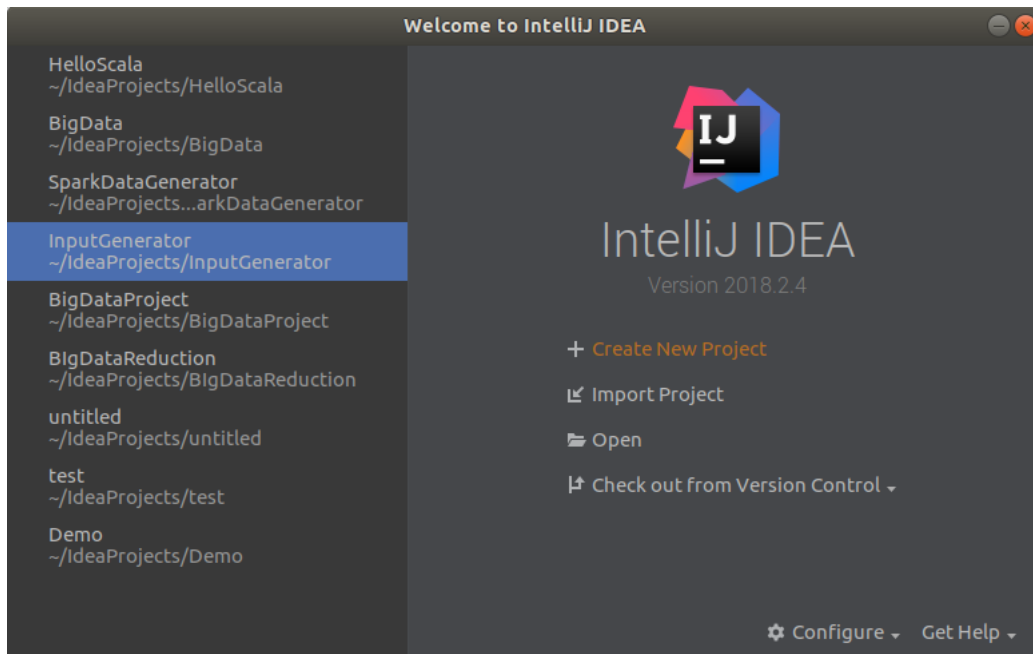
1. Instal dan sudah melakukan konfigurasi untuk Scala, Java, dan Spark.
2. Instal IntelliJ IDEA dari <https://www.jetbrains.com/idea/>.
3. Install `sbt`, berikut adalah langkah instalasi `sbt`:

```
$ echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.list.d/sb
```

```
1 $ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 2EE0EA64E40A89B84B2  
2 $ sudo apt-get update  
3 $ sudo apt-get install sbt
```

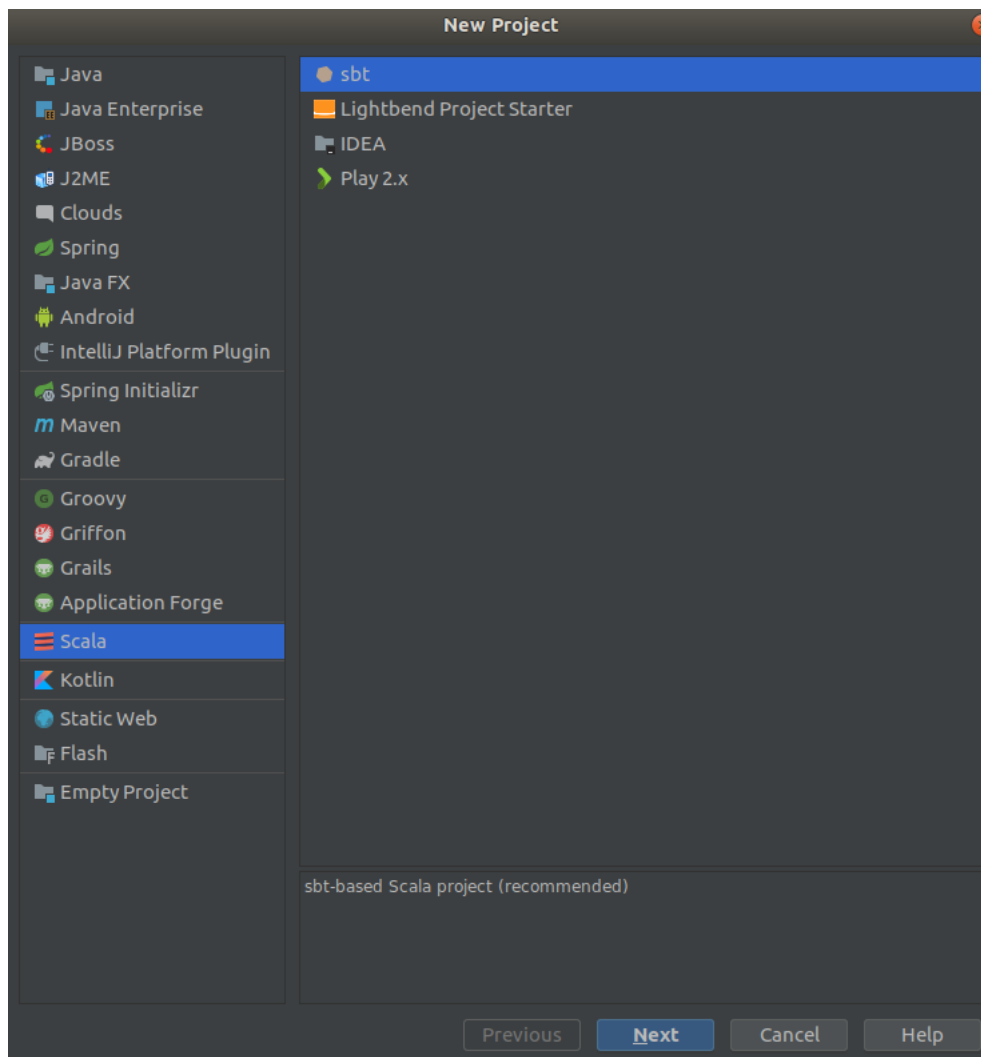
4 Setelah kebutuhan telah terpenuhi maka percobaan bisa dimulai. Berikut adalah langkah-langkah
5 percobaan:

- 6 1. Pertama, buka IntelliJ dan buatlah sebuah project SBT seperti pada Gambar 3.3.



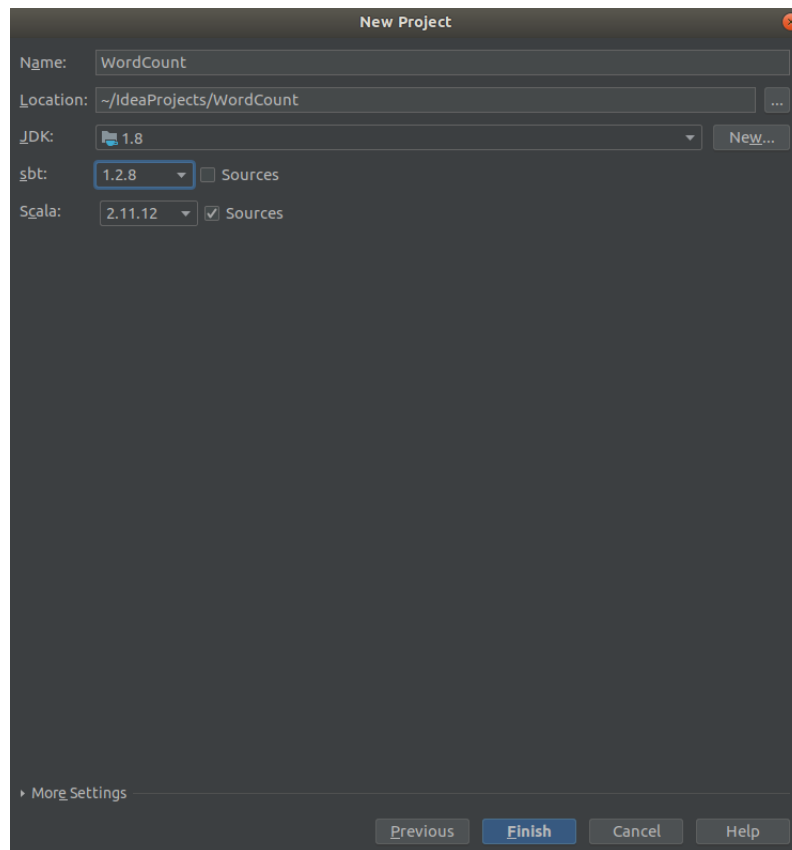
Gambar 3.3: ItelliJ IDEA

- 7 Setelah itu, pilih proyek Scala yang menggunakan sbt. Tekan tombol *next* seperti pada Gambar 3.4.



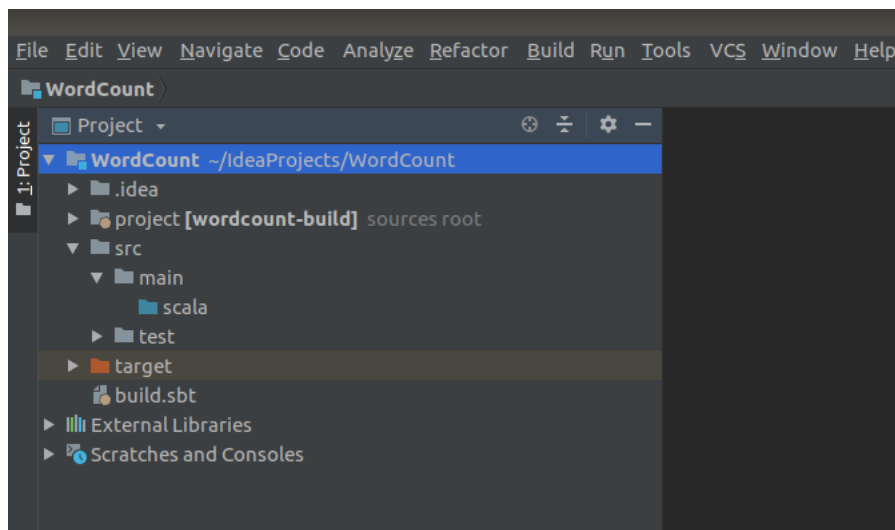
Gambar 3.4: Proyek sbt

- 1 Kemudian, namakan proyek dengan nama WordCount dan pilih versi Sbt, Java, dan Scala yang sesuai
- 2 seperti pada Gambar 3.5.



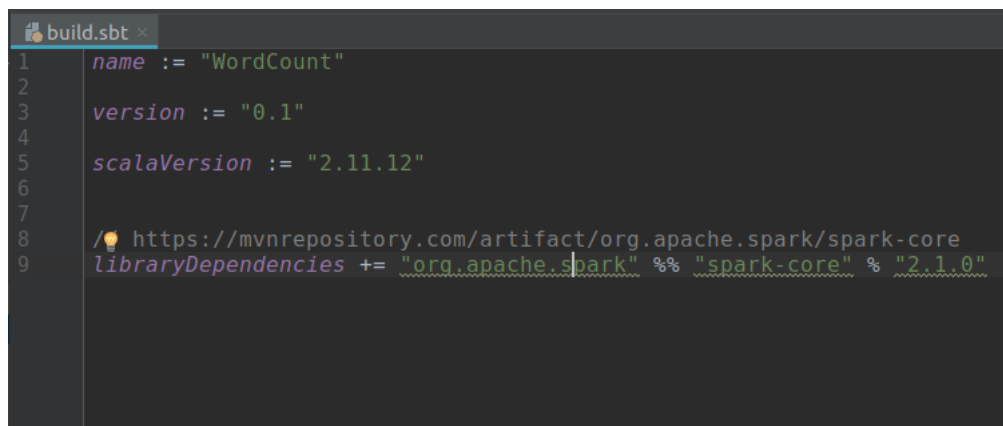
Gambar 3.5: Konfigurasi proyek

- 1 Hasil dari pembuatan proyek baru pada IntelliJ akan terlihat seperti pada Gambar 3.6.



Gambar 3.6: Struktur proyek

- 2 Setelah membuat proyek baru, buka file build.sbt dan tambahkan baris seperti pada Gambar 3.7.



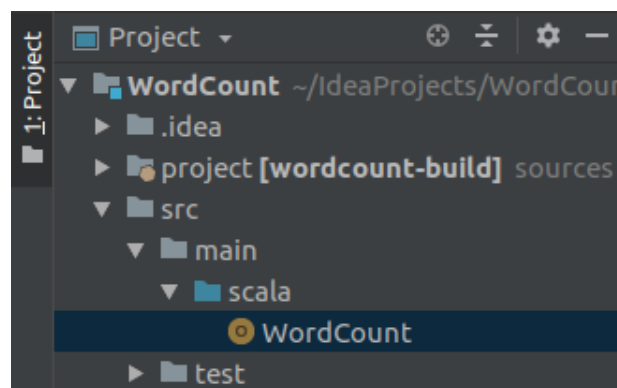
```

1  name := "WordCount"
2
3  version := "0.1"
4
5  scalaVersion := "2.11.12"
6
7
8  // https://mvnrepository.com/artifact/org.apache.spark/spark-core
9  libraryDependencies += "org.apache.spark" %% "spark-core" % "2.1.0"

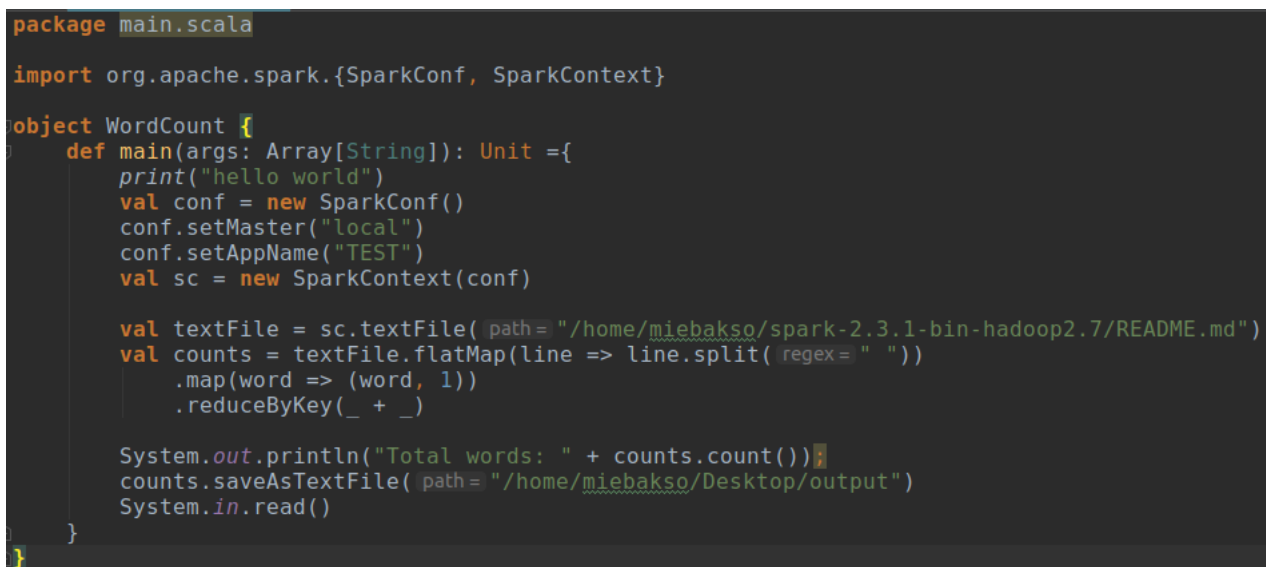
```

Gambar 3.7: Konfigurasi sbt

- 1 3. Tambahkan *object* WordCount pada proyek seperti pada Gambar 3.8.

Gambar 3.8: *object* WordCount

- 2 Setelah itu, tambahkan kode berikut seperti pada Gambar 3.9.



```

package main.scala

import org.apache.spark.{SparkConf, SparkContext}

object WordCount {
  def main(args: Array[String]): Unit = {
    print("hello world")
    val conf = new SparkConf()
    conf.setMaster("local")
    conf.setAppName("TEST")
    val sc = new SparkContext(conf)

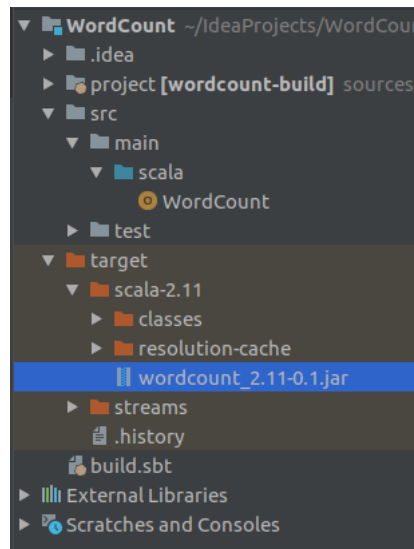
    val textFile = sc.textFile(path = "/home/miebakso/spark-2.3.1-bin-hadoop2.7/README.md")
    val counts = textFile.flatMap(line => line.split(regex = " "))
      .map(word => (word, 1))
      .reduceByKey(_ + _)

    System.out.println("Total words: " + counts.count());
    counts.saveAsTextFile(path = "/home/miebakso/Desktop/output")
    System.in.read()
  }
}

```

Gambar 3.9: Kode WordCount

4. Jalankan perintah 'sbt package' untuk meng-*compile* kode menjadi *executable* JAR seperti pada Gambar 3.10, hasil output dapat dilihat di Gambar 3.11.



Gambar 3.10: JAR

```
miebakso@black:~/IdeaProjects/WordCount$ sbt package
[info] Loading settings for project global-plugins from idea.sbt ...
[info] Loading global plugins from /home/miebakso/.sbt/1.0/plugins
[info] Loading project definition from /home/miebakso/IdeaProjects/WordCount/project
[info] Loading settings for project wordcount from build.sbt ...
[info] Set current project to WordCount (in build file:/home/miebakso/IdeaProjects/WordCount/)
[info] Compiling 1 Scala source to /home/miebakso/IdeaProjects/WordCount/target/scala-2.11/classes ...
[info] Done compiling.
[info] Packaging /home/miebakso/IdeaProjects/WordCount/target/scala-2.11/wordcount_2.11-0.1.jar ...
[info] Done packaging.
[success] Total time: 3 s, completed Apr 17, 2019 4:26:30 PM
miebakso@black:~/IdeaProjects/WordCount$
```

Gambar 3.11: Hasil perintah 'sbt package'

5. Setelah berhasil membuat JAR, kita akan memasukan file JAR kepada *spark-submit* seperti pada Gambar 3.12. Berikut adalah perintah yang harus dijalankan:

```
$ cd $SPARK_HOME
$ ./bin/spark-submit --class main.scala.WordCount --master local[1] \
/home/miebakso/IdeaProjects/WordCount/target/scala-2.11/wordcount_2.11-0.1.jar
```

```
miebakso@black:~/spark-2.3.1-bin-hadoop2.7$ ./bin/spark-submit --class main.scala.WordCount
--master local[2] /home/miebakso/IdeaProjects/WordCount/target/scala-2.11/wordcount_2.11-0.1.jar
2019-04-17 16:47:50 WARN Utils:66 - Your hostname, black resolves to a loopback address:
127.0.1.1; using 192.168.177.101 instead (on interface wlp5s0)
2019-04-17 16:47:50 WARN Utils:66 - Set SPARK_LOCAL_IP if you need to bind to another address
```

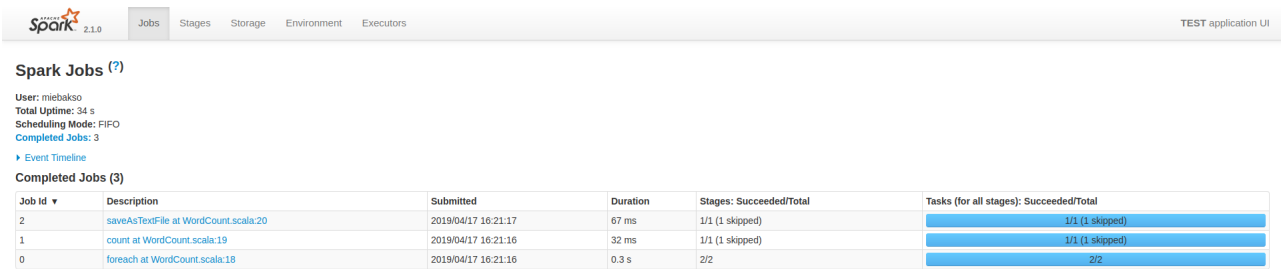
Gambar 3.12: Penggumpulan JAR kepada *spark-submit*

- 1
- Hasil tahap-tahap proses dari program dapat dilihat pada Spark UI dengan membuka alamat yang
- 2
- digaris bawah biru pada Gambar 3.13

```
SparkEnv: Registering MapOutputTracker
SparkEnv: Registering BlockManagerMaster
BlockManagerMasterEndpoint: Using org.apache.spark.storage.DefaultTopologyMapper
BlockManagerMasterEndpoint: BlockManagerMasterEndpoint up
DiskBlockManager: Created local directory at /tmp/blockmgr-4c6caad2-7b7d-42ad-
MemoryStore: MemoryStore started with capacity 1951.2 MB
SparkEnv: Registering OutputCommitCoordinator
Utils: Successfully started service 'SparkUI' on port 4040.
SparkUI: Bound SparkUI to 0.0.0.0, and started at http://192.168.177.101:4040
Executor: Starting executor ID driver on host localhost
Utils: Successfully started service 'org.apache.spark.network.netty.NettyBlock
NettyBlockTransferService: Server created on 192.168.177.101:41353
BlockManager: Using org.apache.spark.storage.RandomBlockReplicationPolicy for
BlockManagerMaster: Registering BlockManager BlockManagerId(driver, 192.168.17
BlockManagerMasterEndpoint: Registering block manager 192.168.177.101:41353 wi
```

Gambar 3.13: Alamat Spark UI

- 3
- Spark UI menggambarkan tahap-tahap proses program. Tampilan dari Spark UI dapat dilihat pada
- 4
- Gambar 3.14.



Gambar 3.14: Spark UI

BAB 4

ANALISIS DAN PERANCANGAN

Pada bab ini, penulis akan menjelaskan apa saja yang dilakukan dalam pengembangan *Agglomerative Hierarchical Clustering* untuk Spark. Pengembangan dilakukan untuk mencapai tujuan yaitu mendapatkan pola dari dataset yang diolah. Pola yang ingin didapatkan meliputi perhitungan rata-rata, nilai maksimum, nilai minimum dan nilai standar deviasi dari setiap atribut yang ada pada data. Selain itu, perlu didapatkan juga jumlah anggota pada setiap *cluster* yang dihasilkan dari algoritma *Hierarchical Agglomerative Clustering*.

4.1 Analisis Masalah

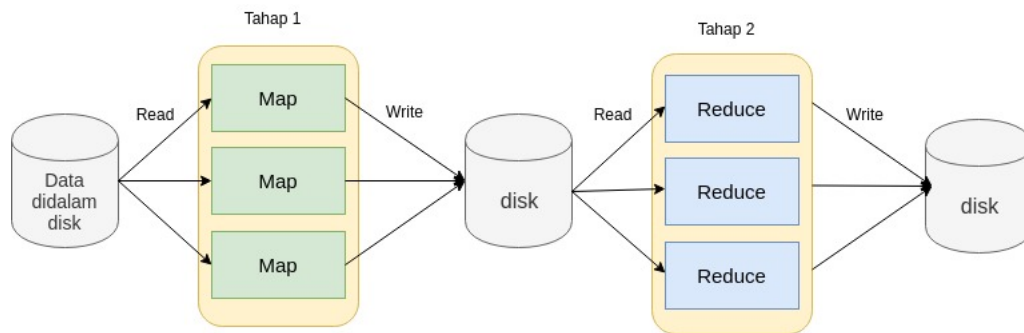
Pada bagian ini akan dijelaskan masalah dari penelitian ini, analisis algoritma *Hierarchical Agglomerative Clustering* dan analisis masukan.

4.1.1 Identifikasi Masalah

Dalam bidang *big data*, volume data yang sangat besar harus disimpan dalam tempat penyimpanan yang sangat besar. Volume data *big data* dapat mencapai *peta bytes*. Volume yang terlalu besar akan meningkatkan biaya dan menghabiskan tempat penyimpanan data. Volume data perlu direduksi agar menghemat tempat dan biaya.

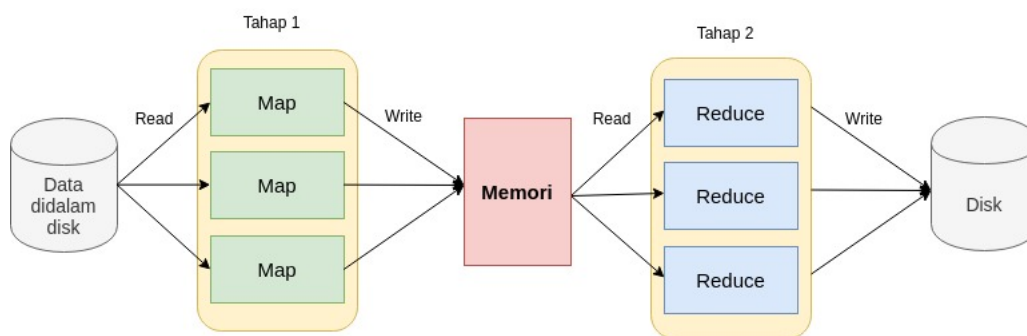
Teknologi Hadoop MapReduce dan algoritma *Agglomerative Hierarchical Clustering* dapat digabungkan sebagai solusi untuk mereduksi data. Algoritma *Agglomerative* dapat mereduksi data dengan mengambil pola-pola dari *clusters* yang dibentuk. Sistem terdistribusi Hadoop membantu dalam proses membagikan dan memecah tugas agar dapat dikerjakan secara paralel. Dengan begitu, proses reduksi data dengan algoritma *Agglomerative* akan lebih cepat.

Tetapi teknologi Hadoop masih terlalu lambat dalam mereduksi data. Hal ini disebabkan karena Hadoop banyak melakukan penulisan dan pembacaan kepada disk. Proses *disk I/O* pada Hadoop sangat tinggi dan menyebabkan algoritma *Agglomerative* berjalan sangat lambat pada Hadoop. Pada setiap tahap Hadoop akan menuliskan hasilnya kepada *disk* dan akan dibaca kembali oleh tahap selanjutnya dari *disk* seperti pada Gambar 4.1.



Gambar 4.1: Penulisan kepada disk di MapReduce

1 Solusinya adalah menggabungkan teknologi terdistribusi lainnya dengan algoritma *Agglomerative* untuk
 2 mereduksi data. Spark, sistem terdistribusi yang menyimpan data pada memori dapat menggantikan Hadoop
 3 MapReduce. Kecepatan memori lebih cepat dibanding disk merupakan salah satu faktor mengapa Spark
 4 akan memproses data dengan kecepatan yang lebih tinggi. Pembacaan dan penulisan akan dilakukan kepada
 5 memori. Gambar 4.2 adalah contoh ilustrasi tahap proses data di Spark.



Gambar 4.2: Penulisan kepada memori di Spark

6 4.1.2 Analisis Hierarchical Agglomerative Clustering MapReduce

7 Sebelum melakukan perancangan, penulis terlebih dahulu mempelajari algoritma *Hierarchical Agglomera-*
 8 *tive Clustering* pada Hadoop. Algoritma *Hierarchical Agglomerative Clustering* pada MapReduce dibagi
 9 menjadi dua bagian. Bagian pertama terkait tahap *map* dan bagian kedua terkait tahap *reduce*. Tahap
 10 *map* bertujuan untuk membagi rata data menjadi beberapa partisi agar setiap *reducer* mendapatkan pekerja-
 11 an yang hampir rata dengan *reducer* yang lainnya. Tahap *map* akan dijelaskan dengan *pseudocode* berikut ini 1:

Algorithm 1: Algoritma *Mapper***Masukan :** Data mentah (**TO**), jumlah partisi (n)**Keluaran :** key = sebuah bilangan bulat $\in \{1 \dots n\}$, $value$ = teks dari sekumpulan nilai atribut yang telah diproses sebelumnya**Deskripsi :** memecah **TO** dengan memberi bilangan acak untuk setiap objek**1 begin****2 value** \leftarrow membaca baris dan memproses atributnya**3 key** \leftarrow sebuah bilangan acak k , dimana $1 \leq k \leq n$ **4 mengembalikan** pasangan $\langle key, value \rangle$ sebagai hasil**5 end**

Tahap *reduce* bertujuan untuk mereduksi data. Pada tahap ini dendrogram akan dibangun dari hasil tahap *map*. Setelah membangun *dendrogram*, *dendrogram* akan dipotong untuk menghasilkan *clusters*. Kemudian, pola akan dihitung dari *clusters* dan disimpan kepada file. Tahap *reduce* akan dijelaskan dengan *pseudocode* berikut ini 2:

Algorithm 2: Algoritma *reducer*

Masukan : pasangan $\langle key, value \rangle$ dari mapper dimana semua *value*-nya memiliki nilai *key* yang sama, *maxObject*, *distType* $\in \{single, complete, means\}$, *cut-off distance* $\{co\}$

Keluaran : pola *cluster*, *c*

Deskripsi : Membuat *dendrogram* dari hasil *map* sesuai dengan batasan yang diberikan, membatasi jumlah objek yang akan diolah menjadi *dendrogram* berdasarkan *maxObject*, menghitung pola dari *cluster* berdasarkan nilai *co*, menuliskan hasil pola kepada file

```

1  begin
2      listTrees  $\leftarrow []$ 
3      foreach pasangan  $\langle key, value \rangle$  do
4          node  $\leftarrow value$ 
5          tambahkan node kepada listTrees
6          isProcessed  $\leftarrow false$ 
7          if listTrees.length == maxObject then
8              bangun dendrogram dari listTrees berdasarkan tipe distType
9              bentuk clusters dari dendrogram berdasarkan nilai co
10             hitung pola c dari setiap cluster yang dibentuk dan simpan hasil kepada file
11             kosongkan listTrees
12             isProcessed  $\leftarrow True$ 
13         end
14     end
15     if isProcessed == false then
16         bangun dendrogram dari listTrees berdasarkan tipe distType
17         bentuk clusters dari dendrogram berdasarkan nilai co
18         hitung pola c dari setiap cluster yang dibentuk dan simpan hasil kepada file
19     end
20 end

```

4.1.3 Analisis Masukan dan Keluaran

Dalam melakukan perancangan perlu diketahui dulu kebutuhan perangkat lunak. Perangkat lunak yang dirancang harus dapat menangani *input* yang diberikan seperti contoh dibawah. Setiap baris mewakili sebuah objek beserta atributnya. Atribut dipisahkan dengan tanda koma. Setiap atribut merupakan bilangan desimal. Setiap objek dapat memiliki lebih dari satu atribut.

```

97.92268076905681,95.67804892782392
15.875897725375477,81.36427207827654
15.825886365695096,6.163384415958262
69.28295038155534,85.36655250595662
10.032110782002924,98.13534474918522
38.53402755308164,96.99987611939603
45.17834148867077,5.96338806209017

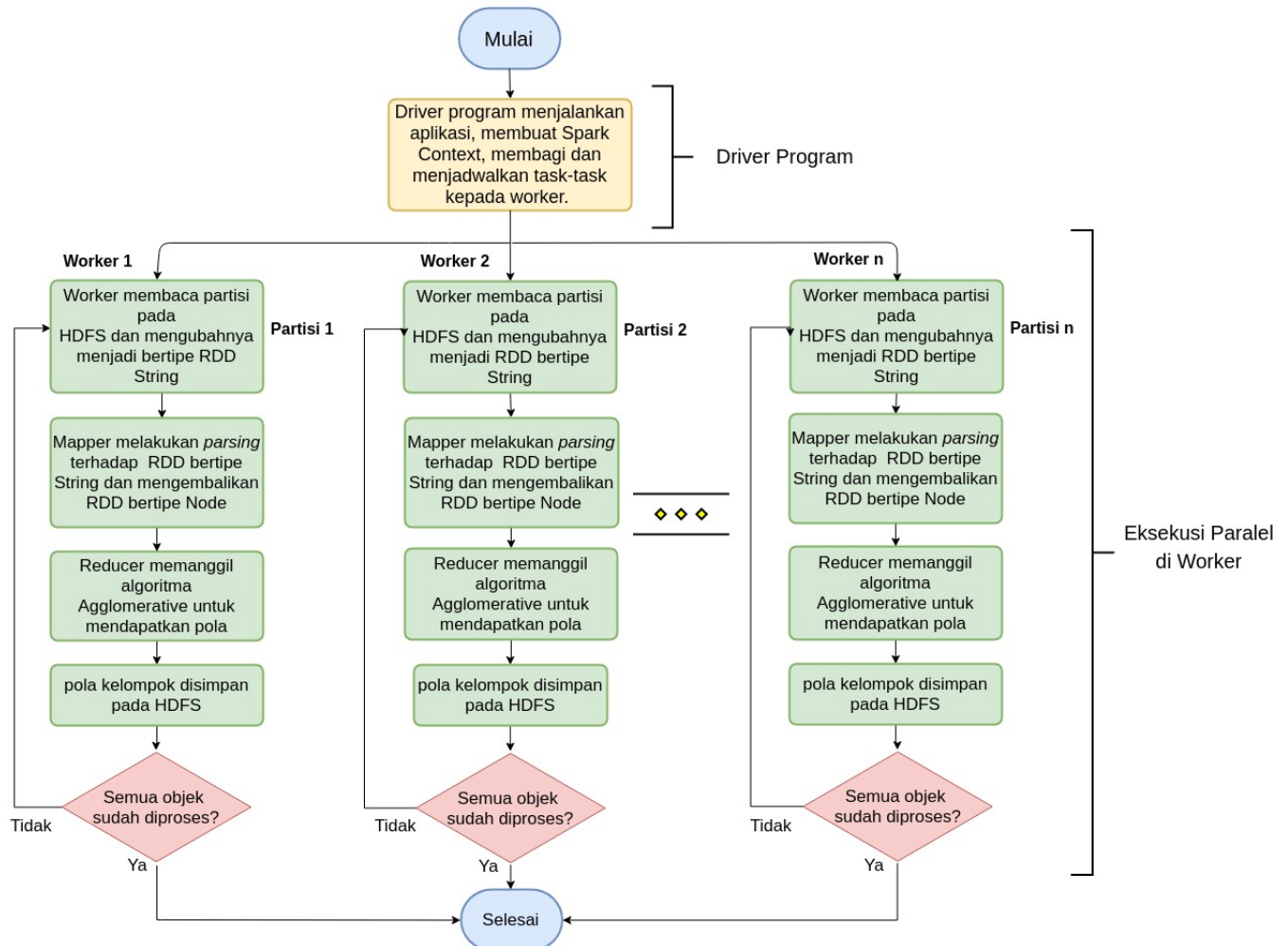
```

1 91.66074344459808,15.182927773314525
 2
 3

4 Selain itu, perangkat lunak harus dapat menghasilkan pola seperti berikut:

- 5 1. Jumlah objek pada *cluster*.
- 6 2. Nilai minimum setiap atribut pada *cluster*.
- 7 3. Nilai maksimum setiap atribut pada *cluster*.
- 8 4. Nilai rata-rata setiap atribut pada *cluster*.
- 9 5. Nilai standar deviasi setiap atribut pada *cluster*.

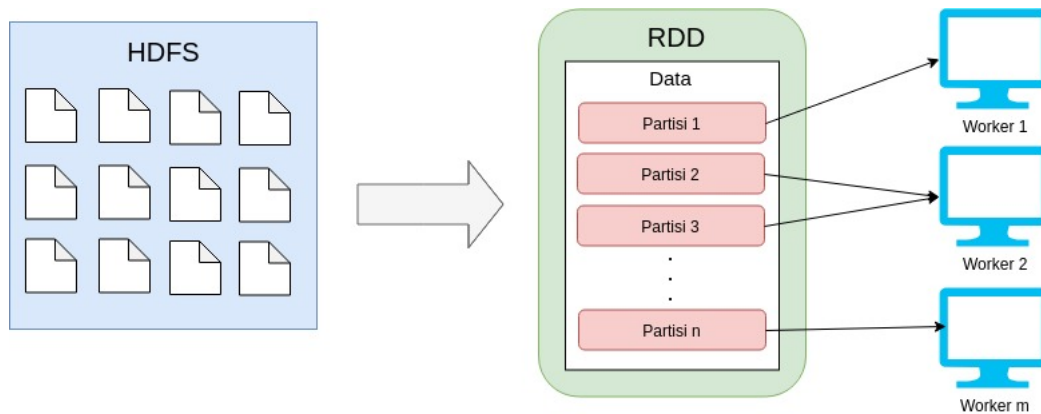
10 4.1.4 Diagram Alur



Gambar 4.3: Diagram alur perangkat lunak

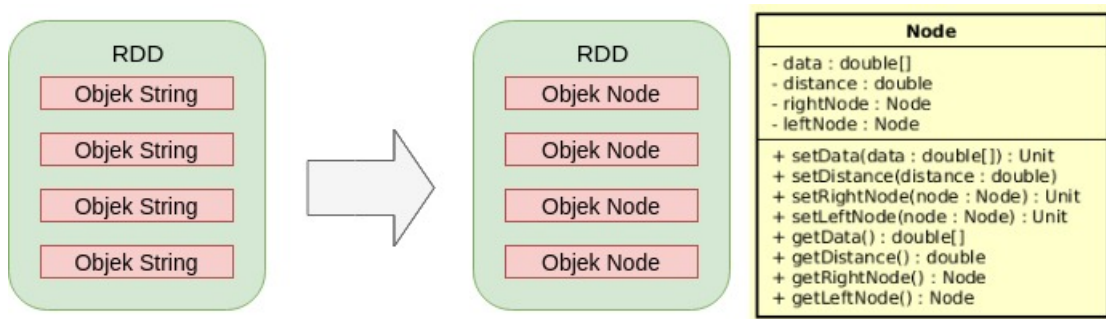
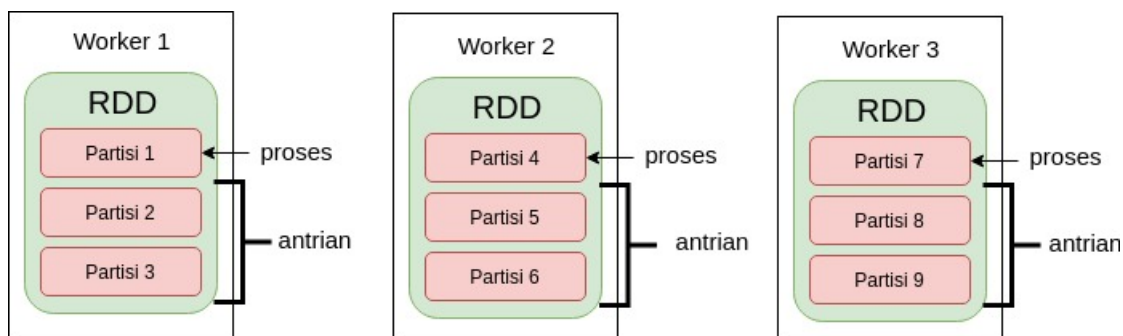
11 Diagram alur diatas (Gambar 4.3) digunakan untuk menjelaskan alur perangkat lunak. Berikut adalah
 12 penjelasan alur perangkat lunak:

1. Pertama-tama aplikasi akan dijalankan pada *driver program*. Kemudian *Spark Context* akan dibuat dan operasi-operasi pada aplikasi diubah menjadi *task-task*. *Task-task* tersebut akan dibagikan dan dijadwalkan kepada *worker* oleh *driver program*.
2. Kemudian, *worker* akan membaca *partisi HDFS* yang ditentukan oleh *driver program*. *Worker* akan mengubah *file-file* tersebut menjadi *RDD* bertipe *String* seperti pada Gambar 4.4.

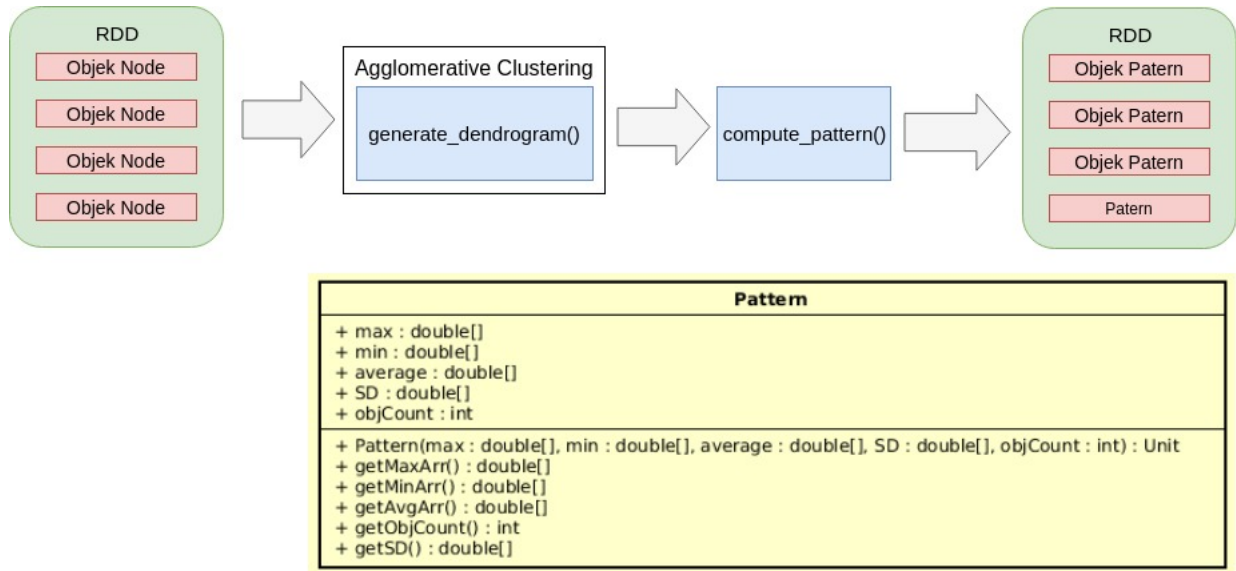


Gambar 4.4: Partisi RDD

3. Selanjutnya, *partisi* yang mengandung *RDD* bertipe *String* akan di-*parsing*. Objek dalam *RDD* bertipe *String* akan diubah menjadi objek *Node* seperti pada Gambar 4.5. Dari tahap ini akan dihasilkan *RDD* berisi objek-objek *Node*. *Worker* akan memproses setiap *partisi* sampai seperti pada Gambar 4.6.

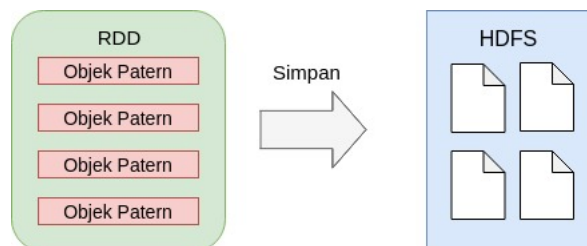
Gambar 4.5: RDD parsing dan kelas *Node*Gambar 4.6: *Worker* memproses partisi

4. Hasil dari tahap map akan dilanjutkan kepada reducer. Pada tahap ini reducer akan memanggil metode *agglomerative clustering* untuk mereduksi keluaran dari mapper seperti pada Gambar 4.7. Pada tahap reducer, akan dibangun *dendrogram* menggunakan algoritma HAC. *Dendrogram* akan dipotong untuk menghasilkan *clusters*. Setiap *cluster* akan dicari polanya, pola ini akan dikembalikan sebagai hasilnya.



Gambar 4.7: Proses reduksi pada *reducer* dan kelas *Pattern*

5. Terakhir, pola dari hasil *reduce* akan disimpan pada *HDFS*. Bila semua partisi sudah diproses, perangkat lunak akan berhenti. Bila masih ada partisi yang tersisa, langkah 3, 4 dan 5 akan diulang kembali sampai semua partisi telah diproses.



Gambar 4.8: Penyimpanan pola pada *HDFS*

4.1.5 Analisis Hierarchical Agglomerative Clustering untuk Spark

Setelah mempelajari algoritma *Hierarchical Agglomerative Clustering* pada MapReduce, format masukan yang harus diproses dan keluaran yang harus dihasilkan, berikut adalah penjelasan *pseudocode* algoritma *map* dan *reduce* untuk Spark:

Algorithm 3: Algoritma *Map***Masukan :** dataset (*D*) bertipe RDD[String]**Keluaran :** *DN* = dataset baru bertipe RDD[Node]**Deskripsi :** Melakukan *parsing* untuk setiap elemen pada RDD *D* dan mengembalikan RDD baru bertipe Node**1 begin****2***DN* \leftarrow RDD bertipe Node yang kosong**3 foreach** *line* pada *D* **do****4***node* \leftarrow node baru**5***split* \leftarrow split *line* berdasarkan delimiter "," dan konversi menjadi double**6**node.setData(*split*)**7***DN* \leftarrow *DN* join *node***8 end****9 return** *DN***10 end**

Algorithm 4: Algoritma *Reduce*

Masukan : (*DN*) RDD[Node] hasil dari mapper, jumlah objek maksimum (*MX*), tipe metode yang dipakai (*distType*) $\in \{single, complete, centroid\}$, dan *cut-off distance* (*co*)

Deskripsi : Membuat *dendrogram* dari hasil *map* sesuai dengan batasan yang diberikan, membatasi jumlah objek yang akan diolah menjadi *dendrogram* berdasarkan *MX*, memotong *dendrogram* berdasarkan nilai *co*, mendapatkan pola *pt* dari potongan *cluster*, menyimpan pola-pola pada HDFS

```

1  begin
2    broadcast nilai MX, distType, dan co
3    objectList  $\leftarrow$  [] array Node kosong
4    patterns  $\leftarrow$  RDD bertipe Pattern untuk mengumpulkan pola hasil reduksi
5    foreach elemen in DN do
6      objectList  $\leftarrow$  objectList join elemen
7      isProcessed  $\leftarrow$  false
8      if count(objectList) == MX then
9        dendrogram  $\leftarrow$  generate_dendrogram(objectList, distType)
10       pt  $\leftarrow$  compute_pattern(dendrogram, co)
11       patterns  $\leftarrow$  pattern join pt
12       isProcessed  $\leftarrow$  true
13       kosongkan objectList
14     end
15   end
16   if isProcessed == false then
17     dendrogram  $\leftarrow$  generate_dendrogram(objectList, distType)
18     pt  $\leftarrow$  compute_pattern(dendrogram, co)
19     patterns  $\leftarrow$  pattern join pt
20   end
21   foreach pattern in patterns do
22     simpan pattern pada HDFS
23   end
24 end

```

Function generate_dendrogram(*objectList*, *distType*):

Masukan : list objek-objek *objectList*, tipe metode *distType*

Keluaran : dendrogram

Deskripsi : Membangun *dendrogram* dari list objek sesuai dengan nilai *distType* yang diberikan

```
1  begin
2      distanceMatrix[][]
3      dendrogram[]
4      inisialisasi distanceMatrix dan dendrogram
5      while dendrogram.length != 1 do
6          gabungkan objek terdekat berdasarkan nilai pada distanceMatrix
7          perbarui dendrogram
8          hitung kembali distanceMatrix berdasarkan distType
9      end
10     return dendrogram[0]
11 end
```

Function `compute_pattern(dendrogram, co):`

Masukan : *dendrogram, cut-off distance co*

Keluaran : pola-pola dari seluruh potongan cluster

Deskripsi : Memotong *dendrogram* menjadi beberapa *clusters* berdasarkan nilai *co*, mendapatkan pola dari setiap *cluster*

```

1  begin
2      bfs[]
3      array kosong bertipe Node
4      clusters[]
5      array kosong untuk menyimpan hasil potongan dari dendrogram
6      bfs.add(dendrogram)
7      dist  $\leftarrow co * dendrogram.distance$ 
8      while bfs tidak kosong do
9          node  $\leftarrow$  bfs.remove(0)
10         if node.distance  $\leq dist$  then
11             clusters.add(node)
12         else
13             left  $\leftarrow$  node.left
14             right  $\leftarrow$  node.right
15             if left  $\neq null$  then
16                 bfs.add(left)
17             end
18             if right  $\neq null$  then
19                 bfs.add(right)
20             end
21         end
22     end
23     patterns[]
24     foreach cluster in clusters do
25         p  $\leftarrow$  dapatkan pola dari setiap cluster
26         patterns.add(p)
27     end
28     return patterns
29 end

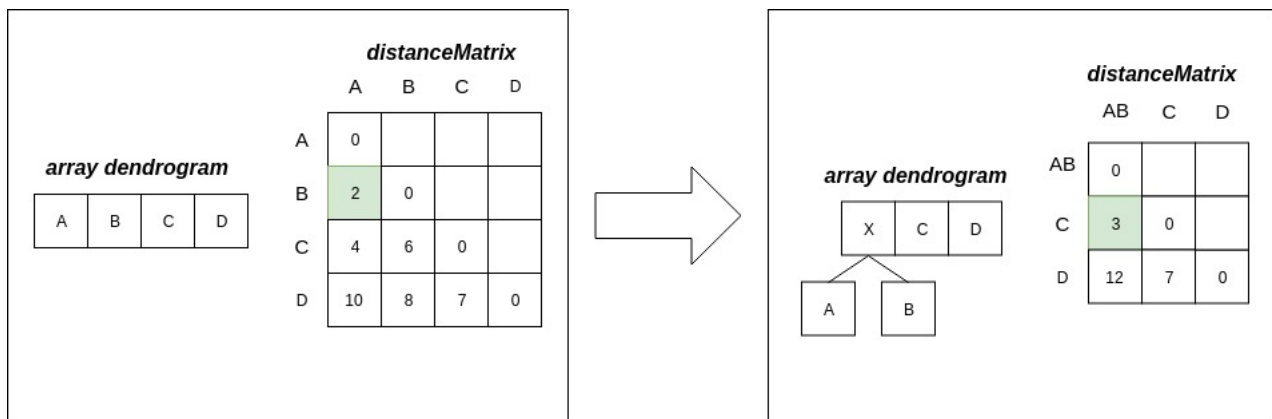
```

Algoritma *Map* 3 ini bertujuan untuk melakukan *parsing* terhadap masukan yang diberikan. Masukan yang masuk berupa RDD[String] akan di-*parsing* menjadi RDD[Node]. Pertama, elemen pada RDD[String] yang berupa *String* akan di pecah berdasarkan *delimiter* "," dan di konversi menjadi bilangan pecahan. Hasilnya merupakan array bertipe *double* yang menjadi atribut objek *Node*. Objek *Node* kemudian akan tambahkan kepada RDD[Node]. RDD[Node] akan dikembalikan sebagai hasil untuk tahap *reduce*.

Algoritma *Reduce* 4 bertujuan untuk membangun dendrogram dan mengembalikan pola-pola bertipe RDD[Pattern] sebagai hasilnya. Pertama-tama nilai *MX*, *distType*, *co* akan di *broadcast* agar setiap *worker*

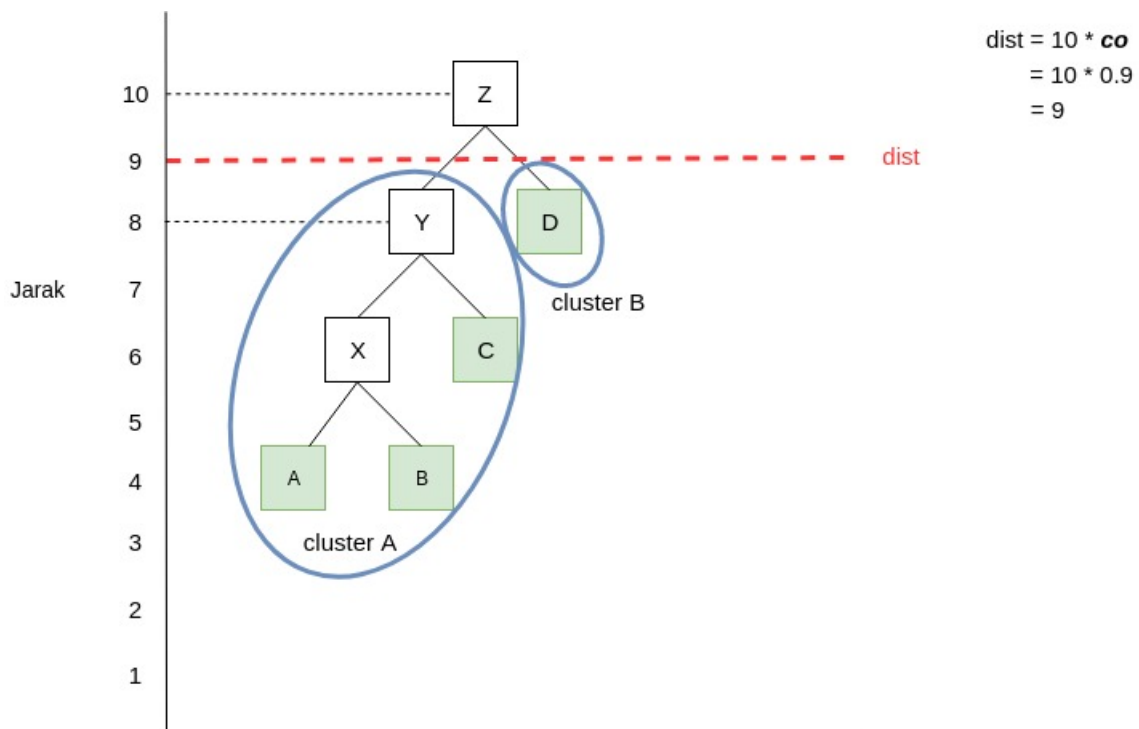
1 memiliki nilai tersebut. *Variable objectList* dibuat untuk menampung *Node-Node* yang akan dibangun
 2 menjadi *dendrogram*. *Node* pada *RDD[Node]* akan ditambahkan kepada *objectList* sampai batas jumlah
 3 *Node* pada *objectList* sama dengan nilai *MX*. Kemudian, fungsi *generate_dendrogram(objectList, distType)*
 4 akan dipanggil untuk membangun dendrogram. Hasil dari fungsi tersebut yaitu sebuah *dendrogram* dijadikan
 5 masukan untuk fungsi *compute_pattern(dendrogram,co)*. Fungsi ini untuk memotong dendrogram menjadi
 6 *clusters* dan mencari pola dari setiap *cluster*. Pola atau *Pattern* akan ditambahkan kepada *variable pattern*
 7 (*RDD[Pattern]*) yang akan dikembalikan sebagai hasil. *variable isProcessed* pada baris 16 untuk memastikan
 8 bahwa setiap elemen pada *objectList* sudah di proses.

10 Fungsi *generate_dendrogram* pada algoritma 4 digunakan untuk membangun *dendrogram*. Fungsi ini
 11 akan menerima *objectList* sebagai masukan. Pertama-tama *array distanceMatrix* harus diinisialisasi dan
 12 dihitung jarak antara objeknya menggunakan *distType* yang ditentukan. Kemudian, *array dendrogram* diisi
 13 dengan objek-objek pada *objectList*. Untuk membangun dendrogram, gabungkan objek yang memiliki nilai
 14 terkecil pada *distanceMatrix* seperti pada Gambar 4.9. Setelah menggabungkan dua buah objek, objek pada
 15 dendrogram akan berkurang 1 dan *distanceMatrix* harus dihitung ulang berdasarkan *distType*.



Gambar 4.9: Contoh perhitungan matriks dan pembentukan dendrogram

16 Fungsi *compute_pattern* pada algoritma 4 digunakan untuk mendapatkan pola dari *cluster*. Fungsi ini
 17 menerima hasil *dendrogram* dari fungsi *generate_dendrogram*, beserta nilai *cut-off distance* sebagai masu-
 18 kannya. Pertama-tama *dendrogram* yang diwakili dengan struktur *tree* akan ditelusuri di setiap tingkatnya.
 19 Jarak pada setiap tingkat akan di cek. Bila jarak sudah kurang dari jarak hasil perkalian *co* dengan tinggi
 20 *dendrogram*, maka *dendrogram* akan dipotong untuk menghasilkan potongan *clusters*. Setelah itu, pola dari
 21 setiap *cluster* akan dicari. Pola didapatkan dengan mencari nilai minimum, maksimum, rata-rata dan standard
 22 deviasi dari setiap attribute pada *cluster*. Pola akan dikembalikan sebagai hasil.

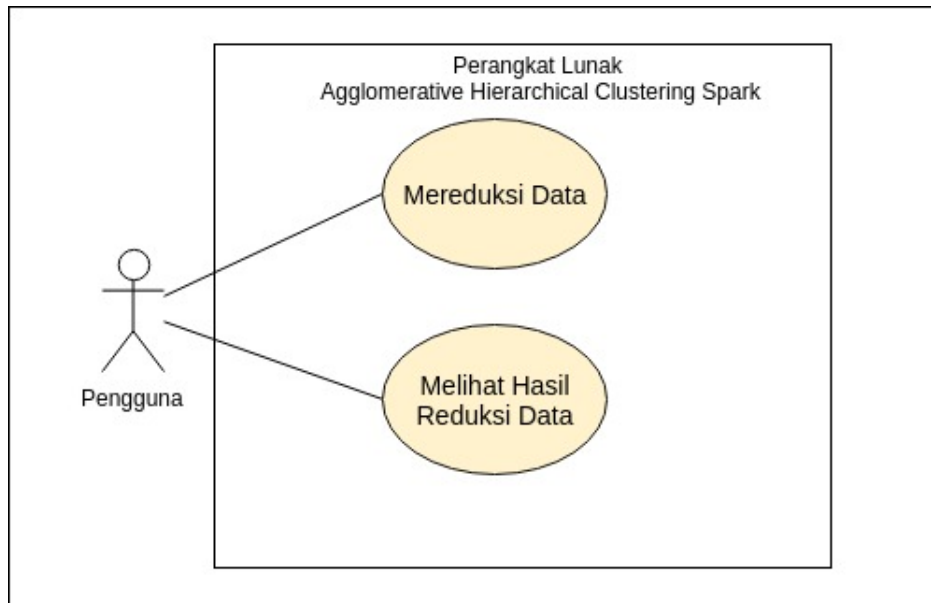
Gambar 4.10: Contoh pemotongan *dendrogram*

4.2 Perancangan Perangkat Lunak

Pada bagian ini, akan dijelaskan perancangan perangkat lunak. Perancangan termasuk diagram *use case*, skenario, diagram kelas, dan rancangan antarmuka.

4.2.1 Diagram Use Case dan Skenario

Diagram *use case* merupakan sebuah pemodelan untuk perilaku dari perangkat lunak yang akan dibuat. Diagram *use case* digunakan untuk mengetahui fungsi apa saja yang ada dalam perangkat lunak. Fungsi-fungsi dari perangkat lunak akan dioperasikan oleh satu pengguna. Cara kerja dan perilaku dari perangkat lunak akan dijelaskan dalam bentuk diagram *use case*. Diagram *use case* dapat dilihat pada Gambar 4.11.



Gambar 4.11: Diagram *use case* perangkat lunak *Hierarchical Agglomerative Clustering*

Berdasarkan gambar diagram *use case* diatas, berikut adalah skenario yang ada:

1. Nama *use case*: Mereduksi data

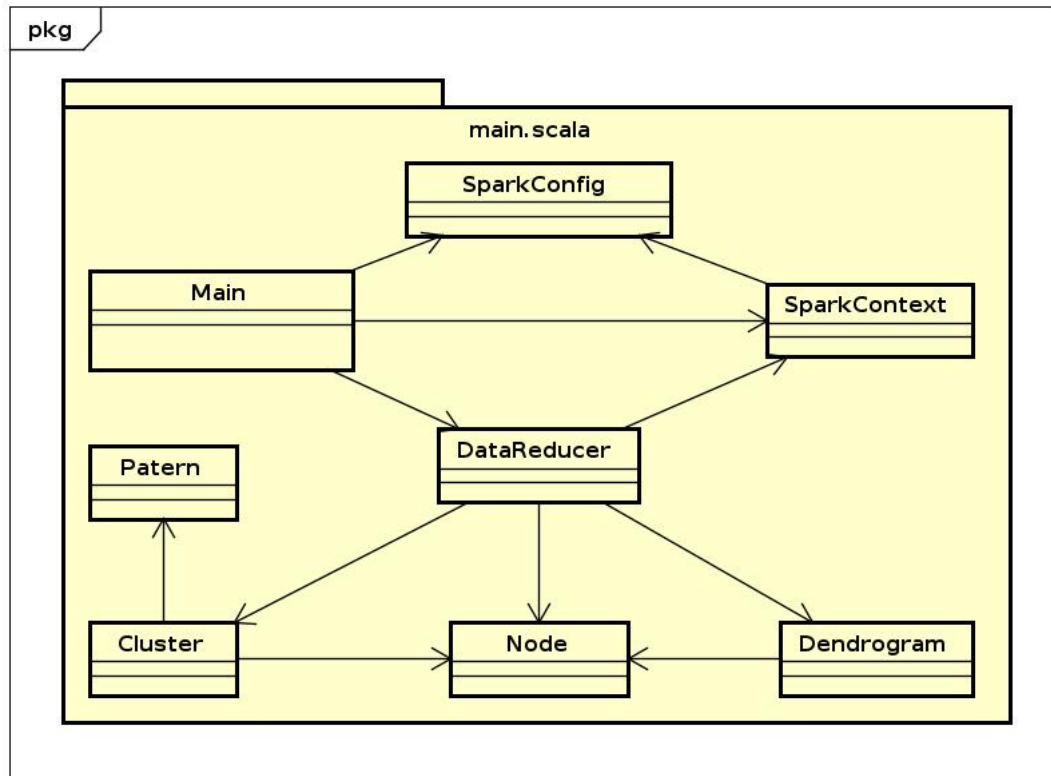
- Aktor: Pengguna
- Pre-kondisi: data yang akan diolah dimasukan kepada HDFS.
- Pra-kondisi: hasil reduksi disimpan pada HDFS.
- Deskripsi: Fitur untuk menjalankan program untuk mereduksi data.
- Langkah-langkah:
 - (a) Pengguna mengisi JAR *path*, *input path*, dan *output path*.
 - (b) Pengguna mengisi jumlah *executor* dan besar *executor memory*.
 - (c) Pengguna mengisi jumlah partisi, batas maksimum objek, tipe metode, dan *cut-off distance*.
 - (d) Pengguna menekan tombol *submit*.
 - (e) Sistem melakukan pengolahan data dengan algoritma *Hierarchical Agglomerative Clustering* pada *cluster Hadoop*.
 - (f) Sistem membuka halaman baru untuk melihat tahap dan progres program.
 - (g) Sistem menyimpan hasil reduksi pada HDFS.

2. Nama *use case*: Mengunduh data

- Aktor: Pengguna
- Pre-kondisi: data yang akan diunduh sudah disimpan pada HDFS.
- Pra-kondisi: data dapat diunduh dari HDFS.
- Deskripsi: fitur untuk mengunduh data hasil reduksi.
- Langkah-langkah:
 - (a) Pengguna mengisi *path* dimana data disimpan pada HDFS.
 - (b) Sistem membuka halaman baru dimana pengguna dapat mengunduh data dari HDFS .

4.2.2 Diagram Kelas

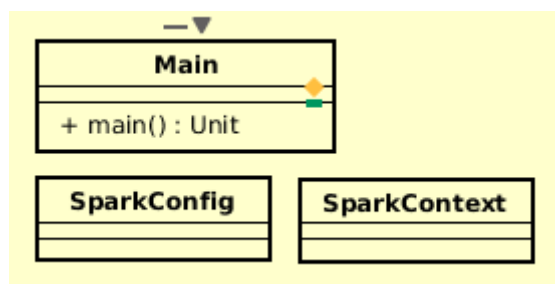
Pada bagian ini akan dijelaskan diagram kelas dari perangkat lunak. Diagram kelas dapat dilihat pada Gambar 4.12.



Gambar 4.12: Diagram kelas

Berdasarkan Gambar 4.12, berikut ini adalah penjelasan kelas-kelas yang digunakan:

- **Main, Spark Config, dan Spark Context**



Gambar 4.13: Kelas Main, SparkConfig, SparkContext

Berikut adalah penjelasan dari ketiga kelas pada Gambar 4.13:

- *Main*: kelas *Main* memiliki *method main* yang merupakan titik masuk dari program. *Method* ini merupakan *method* pertama yang akan dieksekusi ketika program dijalankan.

- *SparkConfig*: kelas *SparkConfig* digunakan untuk mengatur konfigurasi untuk Spark. Pengaturan nama aplikasi, jumlah *core*, besar *memory*, dan lainnya dapat diatur pada kelas ini.
- *SparkContext*: kelas ini merupakan titik masuk untuk layanan-layan dari Apache Spark.

• DataReducer

DataReducer
<pre> + sc : SparkContext + numPar : int + maxObj : int + distanceType : int + cutOffDistance : int + inputPath : int + outputPath : int + DataReducer(sc : SparkContext, nPar : int, maxObject : int, distanceType : int, cutOffDistance : double, inputPath : String, outputPath : String) : Unit + reduceData() : Unit - loadData() : RDD<String> - mapData() : RDD<Node> </pre>

Gambar 4.14: Kelas DataReducer

Kelas *DataReducer* dirancang untuk memproses data. Proses reduksi secara parallel dilakukan pada kelas ini. Proses pemuatan dan penyimpanan data dilakukan pada kelas ini. Berdasarkan Gambar 4.14, berikut adalah penjelasan dari *methods* pada kelas *DataReducer*:

- *loadData*: *method* untuk memuat data berdasarkan *input path* yang diberikan.
- *mapData*: *method* untuk mengubah baris-baris attribut bertipe *String* menjadi objek *Node*. *Method* ini akan mengembalikan RDD bertipe *Node*.
- *reduceData*: *method* untuk mereduksi data menggunakan *agglomerative clustering*. *Method* ini akan mengembalikan pola-pola dari setiap *clusters*.

• Dendrogram

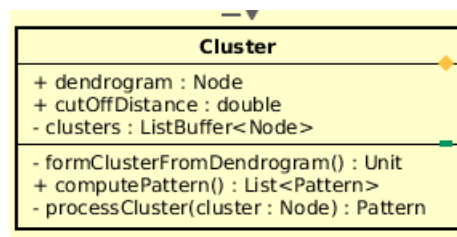
Dendrogram
<pre> - dendrogram : ArrayBuffer<Node> - nodeListCustom : ArrayBuffer<ListBuffer<Node>> - distanceMatrix : ArrayBuffer<ArrayBuffer<Node>> + nodeList : ListBuffer<Node> + distType : int + Dendrogram(nodeList : ListBuffer<Node>, distType : int) : Unit + getDendrogram() : Node + generateDendrogram() : Unit + formClusterBetweenNearestNeighbour() : Unit + recalculateMatrix() : Unit + findMinimumDistance() : Unit + calculateAverageLinkage() : Unit + calculateSingleLinkage() : Unit + calculateAverageLinkage() : Unit </pre>

Gambar 4.15: Kelas Dendrogram

Kelas *Dendrogram* dirancang untuk memproses data dan membangun *dendrogram* sesuai algoritma *Hierarchical Agglomerative Clustering*. Berdasarkan Gambar 4.15, berikut adalah penjelasan *methods* pada kelas *Dendrogram*:

- *getDendrogram*: *method* ini mengembalikan *dendrogram*.
- *generateDendrogram*: *Method* untuk membangun *dendrogram* berdasarkan algoritma *Hierarchical Agglomerative Clustering*.
- *formClusterBetweenNearestNeighbour*: *method* untuk menggabungkan *cluster* terdekat.
- *recalculateMatrix*: *method* untuk menghitung ulang matriks jarak.
- *findMinimumDistance*: *method* untuk mencari jarak minimum antara dua *cluster*.
- *calculateCentroidLinkage*: *method* untuk mencari jarak antara *centorid* dua buah *cluster*.
- *calculateSingleLinkage*: *method* untuk mencari jarak minimum antara dua buah *cluster*.
- *calculateCompleteLinkage*: *method* untuk mencari jarak maksimum antara dua *cluster*.
- *calculateDistance*: *method* untuk mencari jarak antara dua buah *Node* berdasarkan atributnya.

• Cluster



Gambar 4.16: Kelas Cluster

Kelas *Cluster* dirancang untuk mengolah *cluster* untuk menghasilkan pola dengan memotong *cluster*. Berdasarkan Gambar 4.16, berikut adalah penjelasan *methods* pada kelas *Cluster*:

- *formClusterFromDendrogram*: *method* ini bertugas untuk memotong *dendrogram* menjadi beberapa *cluster*.
- *computePattern*: *method* untuk mengolah potongan-potongan *cluster* menjadi pola dengan memanggil *method* *processCluster*.
- *processCluster*: *method* untuk memproses *cluster* dan membuat pola berdasarkan anggota-anggota pada *cluster*.

• Pattern

Pattern
+ max : double[] + min : double[] + average : double[] + SD : double[] + objCount : int
+ Pattern(max : double[], min : double[], average : double[], SD : double[], objCount : int) : Unit + getMaxArr() : double[] + getMinArr() : double[] + getAvgArr() : double[] + getObjCount() : int + getSD() : double[]

Gambar 4.17: Kelas Pattern

Kelas *Pattern* dirancang untuk merepresentasikan pola pada *cluster*. Berdasarkan Gambar 4.16, berikut adalah penjelasan *methods* pada kelas *Pattern*:

- *getMaxArr*: *method* ini mengembalikan *array* berisi nilai maksimum dari setiap atribut.
- *getMinArr*: *method* ini mengembalikan *array* berisi nilai minimum dari setiap atribut.
- *getAvgArr*: *method* ini mengembalikan *array* berisi nilai rata-rata dari setiap atribut.
- *getSDArr*: *method* ini mengembalikan *array* berisi nilai standar deviasi dari setiap atribut.
- *getObjCount*: *method* ini mengembalikan jumlah objek.

• Node

Node
- data : double[] - distance : double - rightNode : Node - leftNode : Node
+ setData(data : double[]) : Unit + setDistance(distance : double) + setRightNode(node : Node) : Unit + setLeftNode(node : Node) : Unit + getData() : double[] + getDistance() : double + getRightNode() : Node + getLeftNode() : Node

Gambar 4.18: Kelas Node

Kelas *Node* digunakan untuk membentuk pohon yang merepresentasikan *dendrogram*. Selain itu kelas ini digunakan untuk merepresentasikan anggota pada *cluster*. Berdasarkan Gambar 4.18, berikut adalah penjelasan *methods* pada kelas *Node*:

- *setData*: *method* untuk memasukan nilai-nilai atribut.
- *setDistance*: *method* untuk megubah nilai jarak.
- *setRightNode*: *method* untuk menambahkan anak kanan *Node*.
- *setLeftNode*: *method* untuk menambahkan anak kiri *Node*.

- *getData: method* ini mengembalikan nilai-nilai atribut.
- *getDistance: method* ini mengembalikan jarak.
- *getRightNode: method* ini mengembalikan anak belah kanan dari *Node*.
- *getLeftNode: method* ini mengembalikan anak belah kiri dari *Node*.

4.2.3 Rancangan Antarmuka

Antarmuka dirancang untuk mempermudah pengguna dalam menjalankan program dan mengambil hasil data yang telah direduksi. Ada dua buah menu utama yang dapat dipilih oleh pengguna, menu *Submit* dan *Data*. Menu *Submit* digunakan untuk menjalankan aplikasi dan menu *Data* digunakan untuk mengunduh data hasil reduksi. Berikut adalah penjelasan rancangan antarmuka:

1. Perancangan halaman *Submit* untuk mempermudah penggunaan menjalankan aplikasi. Pada halaman ini, disediakan *form* beserta *input parameter* yang dibutuhkan untuk menjalankan aplikasi. Gambar rancangan antarmuka dapat dilihat pada Gambar 4.19

The screenshot shows a web browser window titled 'Spark UI Page' with the address bar showing 'localhost/sparkUI/index.php'. The page has a navigation bar with 'Submit' and 'Patern' (likely a typo for Pattern) tabs. The main content area is titled 'Submit Spark Application' and contains a form with the following fields:

- Spark JAR Path (text input)
- input path (text input)
- output path (text input)
- number of executor (text input)
- executor memory (text input)
- number of partition (text input)
- max object (text input)
- single linkage (dropdown menu)
- complete linkage (dropdown menu)
- centroid linkage (dropdown menu)
- cut off distance (text input)
- submit (button)

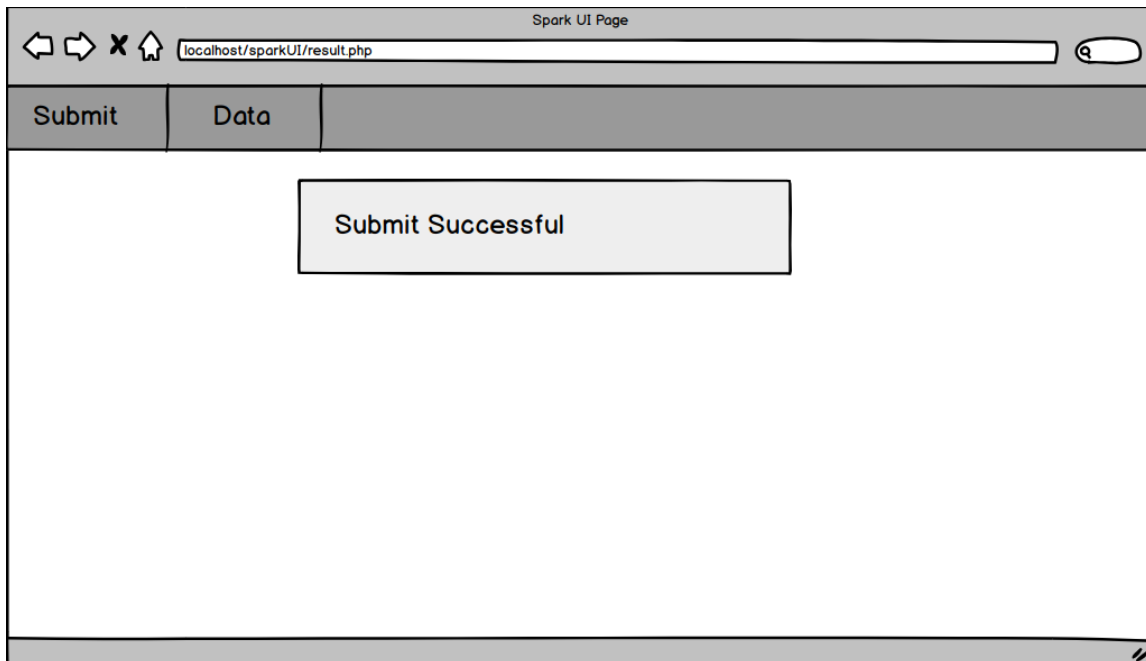
Gambar 4.19: Rancangan antarmuka menu *submit*

Berdasarkan Gambar 4.19, berikut adalah penjelasan *input field* yang ada:

- *Spark JAR Path: field* untuk direktori JAR.
- *input path: field* untuk direktori file *input* pada HDFS.
- *output path: field* untuk direktori tempat penyimpanan hasil pada HDFS.
- *number of executor: field* untuk menentukan jumlah *executor* yang akan dipakai.
- *executor memory: field* untuk menentukan jumlah memori yang akan dipakai.
- *number of partition: field* untuk menentukan jumlah partisi untuk data.

- *max object: field* untuk membatasi jumlah objek pada yang akan diolah.
- *drop down (single linkage, complete linkage, centroid linkage)*: kotak pilihan untuk memilih metode *single linkage*, *complete linkage* atau *centroid linkage* yang digunakan untuk memproses data.
- *cut off distance: field* untuk menentukan jarak untuk memotong *dendrogram* menjadi *clusters*.

Setelah pengguna menekan tombol *submit* pada *form*, pengguna akan dipindahkan ke halaman baru seperti pada Gambar 4.20. Selain itu, halaman *Hadoop web UI* akan terbuka di tab baru seperti pada Gambar 4.21.



Gambar 4.20: Rancangan antaramuka halaman *result*



All Applications

Cluster

- About
- Nodes
- Node Labels
- Applications
- NEW
- NEW SAVING
- SUBMITTED
- ACCEPTED
- RUNNING
- FINISHED
- FAILED
- KILLED
- Scheduler

Tools

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory
1	0	0	1	0	0 B	3 GB

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes
2	0	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation
Capacity Scheduler	[MEMORY]	<memory:128, vCores:1>

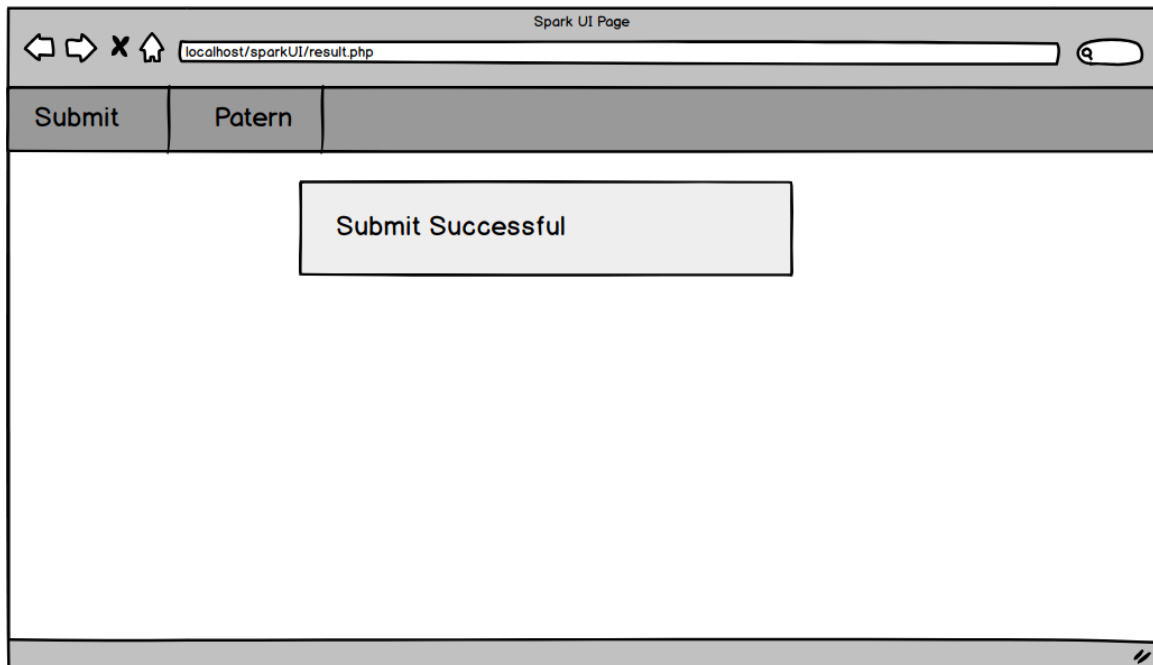
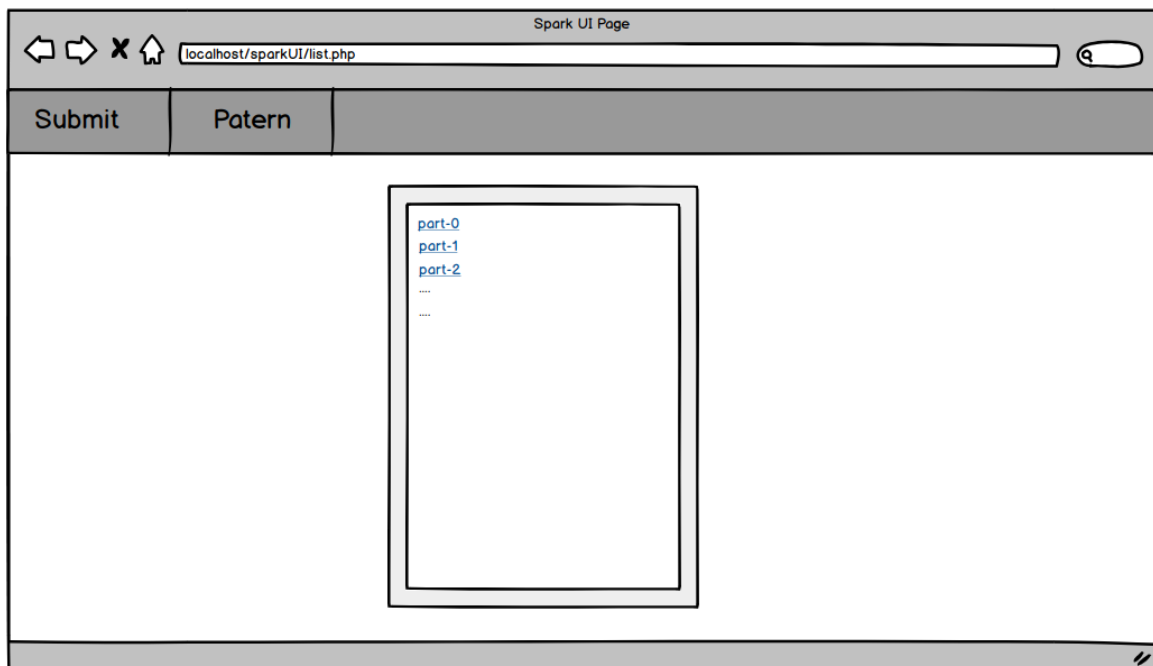
Show 20 entries

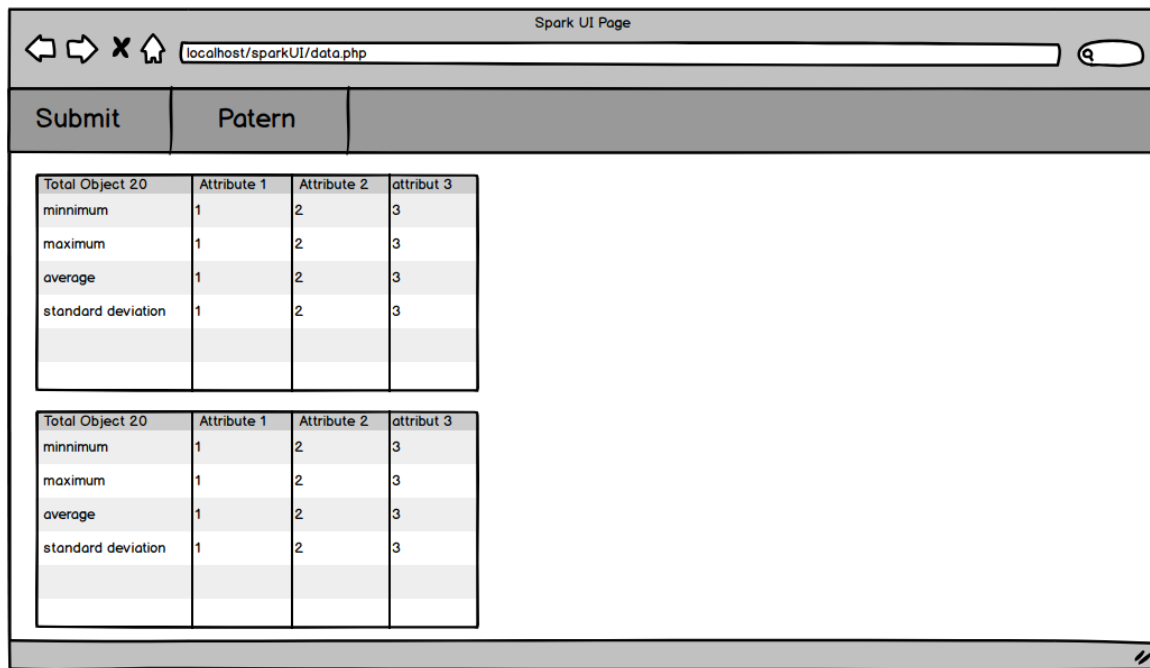
ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus
application_1507721641254_0001	hadoop	word count	MAPREDUCE	default	0	Wed Oct 11 13:39:33 +0200 2017	Wed Oct 11 13:40:13 +0200 2017	FINISHED	SUCCEEDED

Showing 1 to 1 of 1 entries

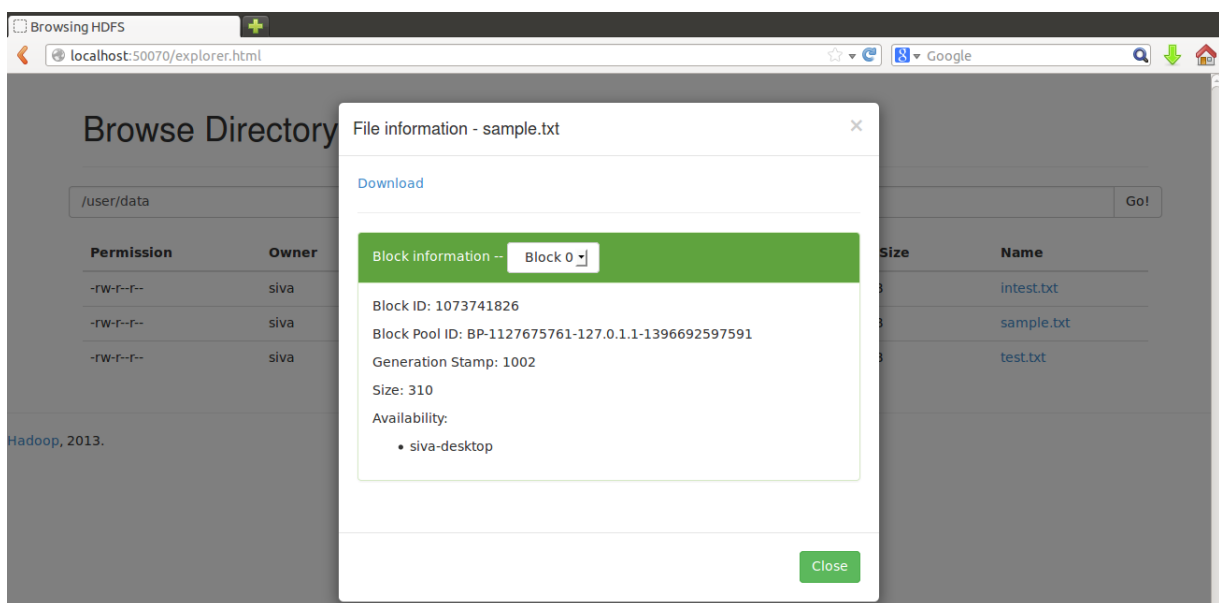
Gambar 4.21: Halaman web Hadoop

- 1 2. Perancangan halaman antarmuka menu *Data* (Gambar 4.22) digunakan untuk membuka direktori
- 2 dimana data disimpan pada HDFS. Ketika pengguna memasukkan direktori, pengguna akan dipindahkan
- 3 ke halaman baru (Gambar 4.23) dan sebuah halaman (Gambar 4.25) akan dibuka untuk menampilkan
- 4 data yang bisa diunduh. Ketika pengguna menekan salah satu nama file pada halaman *list* (4.23),
- 5 pengguna dapat melihat isi dari data yang ada dari file tersebut di halaman (4.24).

Gambar 4.22: Rancangan antarmuka menu *Data*Gambar 4.23: Rancangan antarmuka halaman *list*



Gambar 4.24: Rancangan antarmuka halaman data



Gambar 4.25: Halaman web HDFS

BAB 5

IMPLEMENTASI DAN PENGUJIAN PERANGKAT LUNAK

5.1 Implementasi Perangkat Lunak

5.1.1 Lingkungan Perangkat Keras

Perangkat keras yang digunakan dalam membangun perangkat lunak adalah sebuah PC dengan spesifikasi berikut:

- *Processor*: Intel i7 4790K @4.00 GHz
- RAM: 16 GB DDR3
- VGA: NVIDIA GeForce GTX 750TI 2GB
- *Harddisk*: 1TB + 256GB SSD

5.1.2 Lingkungan Perangkat Lunak

Perangkat lunak yang digunakan untuk membangun perangkat lunak adalah sebagai berikut:

- Sistem Operasi: Ubuntu 18.04.2 LTS
- Bahasa Pemrograman: Scala
- IDE: IntelliJ IDE 2018
- Versi Java: JDK 1.8.0_181
- Versi Scala: Scala 2.11.12
- Versi SBT: SBT 1.2.8
- Library Dependency:
 - org.apache.spark:spark-core 2.1.0
 - org.scala-lang:scala-library 2.11.12

5.1.3 User Interface

Implementasi rancangan tampilan antarmuka pada perangkat lunak ini menggunakan html,css, dan php. Berikut adalah tampilan setiap halaman:

1. Implementasi antarmuka untuk menu *Submit* dapat dilihat pada Gambar 5.1.

Submit Patern

Submit Spark Application

Spark JAR Path

Input Path

Output Path

Number of Executor

1

Executor Memory in mb

1000

Number of Partition

1

Max Object

1

Single Linkage

Cut Off Distance

0.1

SUBMIT

Gambar 5.1: Tampilan menu *submit*

2. Implementasi antarmuka untuk menu *Data* dapat dilihat pada Gambar 5.2.

Submit Patern



Explore HDFS Directory

HDFS data path

SUBMIT

Gambar 5.2: Tampilan menu *Data*

- 1 3. Implementasi antarmuka sesudah melakukan *submit* dapat dilihat pada Gambar 5.3.

Submit Patern

Submit Successful

Gambar 5.3: Tampilan halaman sesudah *submit*

- 2 4. Implementasi antarmuka halaman *list* dapat dilihat pada Gambar 5.4.

Submit Patern

```
part-0  
part-1  
part-2
```

Gambar 5.4: Tampilan halaman *list*

- 1 5. Implementasi antarmuka halaman *data* dapat dilihat pada Gambar [5.5](#).

Submit Patern		
Total Obj = 13	attribute-1	attribute-2
Minimum	2.989196145255757	32.668909774235246
Maximum	58.86655003746121	90.87631433387419
Average	32.87713041936097	65.42538643824919
Standard Deviation	14.941607927042684	17.27969410300335

Total Obj = 5	attribute-1	attribute-2
Minimum	1.715184579886253	6.548756714677017
Maximum	25.99400990511951	26.14315566259172
Average	16.39405819212848	17.238589783987994
Standard Deviation	11.24161321121706	7.833594989767486

Total Obj = 5	attribute-1	attribute-2
Minimum	48.66269805215405	5.134186149105857

Gambar 5.5: Tampilan halaman data

5.2 Pengujian Fungsional Perangkat Lunak

Perangkat lunak yang disusun oleh penulis telah diuji untuk membuktikan kebenaran dari perangkat lunak. Program akan dieksekusi dan kemudian diamati apakah hasil sesuai dengan yang diinginkan. Perangkat lunak akan diberikan data dengan ukuran yang kecil beserta *parameter* yang sudah ditentukan.

- Pada percobaan pertama, akan digunakan metode *single linkage*, dengan jumlah partisi = 1, jumlah objek maksimum = 4, dan nilai *cut-off distance* = 0.8. Berikut adalah data yang digunakan untuk pengujian:

4.0, 5.0

1 3.0,7.0
 2 4.0,3.0
 3 10.0,7.0
 4 10.0,10.0

5 Hasil dari percobaan pertama adalah sebagai berikut:

6 3
 7 3.0,3.0
 8 4.0,7.0
 9 3.6666666666666665,5.0
 10 0.5773502691896258,2.0
 11 1
 12 10.0,7.0
 13 10.0,7.0
 14 10.0,7.0
 15 0.0,0.0
 16 1
 17 10.0,10.0
 18 10.0,10.0
 19 10.0,10.0
 20 0.0,0.0

21 • Pada percobaan kedua, akan digunakan metode *complete linkage*, dengan jumlah partisi = 1, jumlah
 22 objek maksimum = 4, dan nilai *cut-off distance* = 0.8. Berikut adalah data yang digunakan untuk
 23 pengujian:

24 4.0,5.0
 25 3.0,7.0
 26 4.0,3.0
 27 10.0,7.0
 28 10.0,10.0

29 Hasil dari percobaan kedua adalah sebagai berikut:

30 3
 31 3.0,3.0
 32 4.0,7.0
 33 3.6666666666666665,5.0
 34 0.5773502691896258,2.0
 35 1
 36 10.0,7.0
 37 10.0,7.0

1 10.0,7.0
 2 0.0,0.0
 3 1
 4 10.0,10.0
 5 10.0,10.0
 6 10.0,10.0
 7 0.0,0.0

8 • Pada percobaan ketiga, akan digunakan metode *centroid linkage*, dengan jumlah partisi = 1, jumlah
 9 objek maksimum = 4, dan nilai *cut-off distance* = 0.8. Berikut adalah data yang digunakan untuk
 10 pengujian:

11 4.0,5.0
 12 3.0,7.0
 13 4.0,3.0
 14 10.0,7.0
 15 10.0,10.0

16 Hasil dari percobaan ketiga adalah sebagai berikut:

17 3
 18 3.0,3.0
 19 4.0,7.0
 20 3.6666666666666665,5.0
 21 0.5773502691896258,2.0
 22 1
 23 10.0,7.0
 24 10.0,7.0
 25 10.0,7.0
 26 0.0,0.0
 27 1
 28 10.0,10.0
 29 10.0,10.0
 30 10.0,10.0
 31 0.0,0.0

32 Berdasarkan hasil ketiga percobaan yang didapat, maka dapat disimpulkan bahwa perangkat lunak
 33 sudah dapat melakukan proses reduksi data menggunakan algoritma *Agglomerative Clustering* berdasarkan
 34 metode yang dipilih dengan benar. Pola yang dihasilkan oleh perangkat lunak sudah sesuai dengan apa yang
 35 diharapkan.

36 5.3 Hasil Eksperimen Perangkat Lunak

37 Pada bagian ini akan diuji performa perangkat lunak Spark dan Hadoop. Kedua perangkat lunak akan diban-
 38 dingkan hasil eksekusi waktunya. Karena perangkat lunak hadoop tidak dapat menghitung standar deviasi,

maka perangkat lunak Hadoop akan dibandingkan dengan perangkat lunak Spark yang tidak menghitung standar deviasi dan yang menghitung standar deviasi. Data yang digunakan pada percobaan merupakan data yang dihasilkan secara acak dengan ukuran yang berbeda-beda. Data-data tersebut memiliki dua atribut bilangan pecahan yang dipisahkan dengan tanda koma. Jumlah objek pada setiap ukuran data dapat dilihat pada Tabel 5.1.

Tabel 5.1: Tabel data yang digunakan pada eksperimen

Ukuran Data	Jumlah Objek	Jumlah Block
1 GB	36000000	40
2 GB	64000000	70
3 GB	81000000	89
5 GB	144000000	157
10 GB	256000000	279
15 GB	400000000	435
20 GB	529000000	576

Berikut adalah spesifikasi perangkat keras yang digunakan:

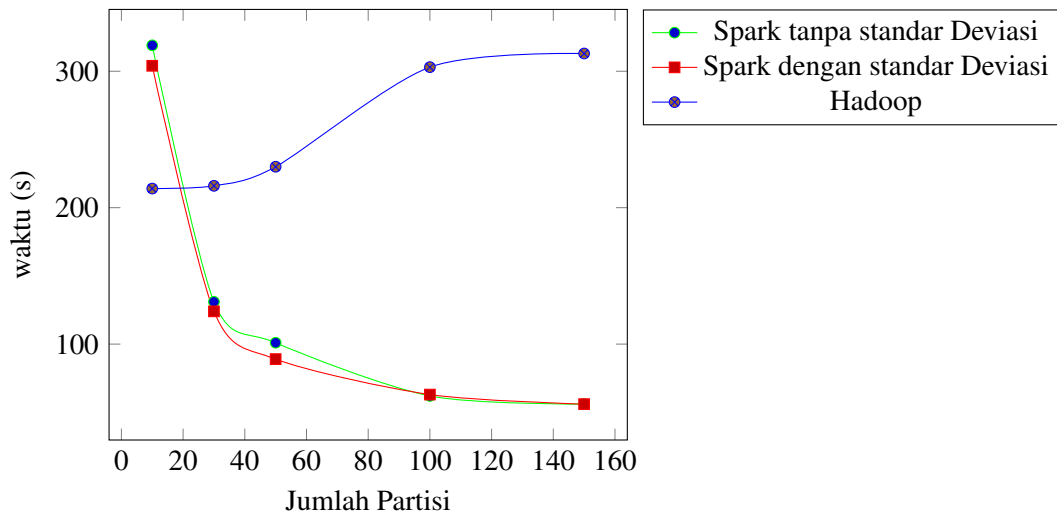
- *Processor*: Intel core i5 8500 @3.00 GHz, 6 core
- RAM: 8GB
- *Harddisk*: 500GB
- Sistem Operasi: Ubuntu 18.0.4

5.4 Percobaan Dampak Partisi pada Performa Perangkat Lunak Spark dan Hadoop

Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang berbeda. Percobaan ini akan menggunakan 1 komputer sebagai komputer *master* dan 10 komputer lainnya sebagai *worker* dengan setiap *worker* menggunakan 1 core. Ukuran data yang digunakan adalah 1 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 30. Tabel (5.2) berikut adalah hasil dari eksperimen:

Tabel 5.2: Percobaan Jumlah Partisi Hadoop dan Spark dengan Ukuran Data 10 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark Tanpa standar Deviasi (Detik)	Waktu Eksekusi Spark (Detik)	Waktu Eksekusi Hadoop (Detik)	Hasil Reduksi Spark Tanpa standar Deviasi (GB)	Hasil Reduksi Spark (GB)	Hasil Reduksi Hadoop (GB)
1	10	319	304	214	0.54	0.67	0.57
1	30	131	124	216	0.54	0.67	0.57
1	50	101	89	230	0.54	0.67	0.57
1	100	62	63	303	0.54	0.67	0.57
1	150	56	56	313	0.54	0.67	0.57



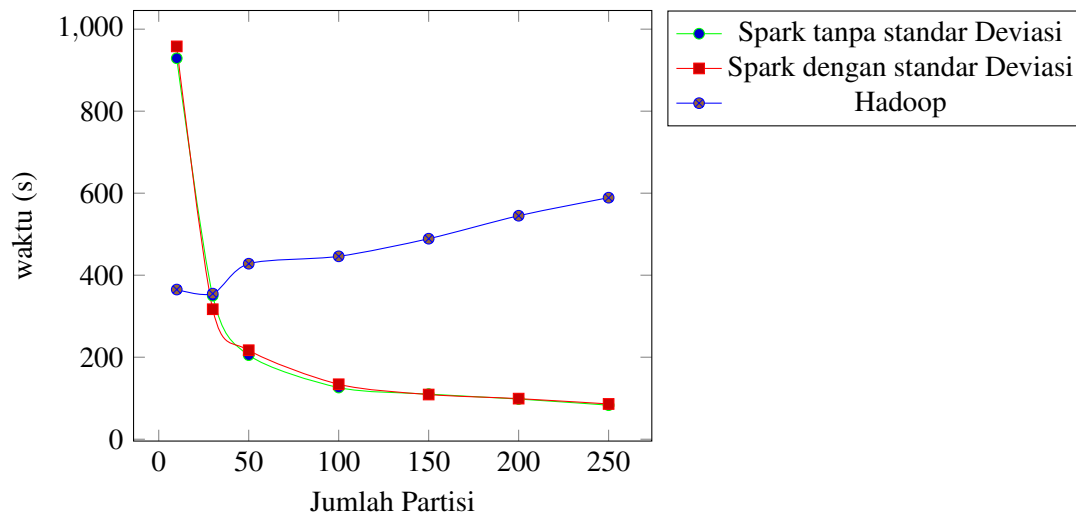
Gambar 5.6: Hasil Percobaan Partisi Spark dan Hadoop 1GB

Berdasarkan hasil grafik (5.6), dapat dilihat bahwa waktu eksekusi Spark menurun dan waktu eksekusi Hadoop meningkat ketika jumlah partisi diperbesar. Waktu eksekusi Hadoop menaik secara konsisten ketika jumlah partisi diperbesarkan. Waktu eksekusi Spark menurun drastis pada awalnya ketika jumlah partisi ditingkatkan sampai titik tertentu dimana peningkatan jumlah partisi tidak memiliki dampak yang sangat drastis pada waktu eksekusi Spark. Tidak ada perbedaan yang jauh antara waktu eksekusi aplikasi Spark dengan standar deviasi maupun yang tidak. Aplikasi Spark memiliki waktu eksekusi yang lebih baik dibanding Hadoop pada jumlah partisi yang besar dan waktu eksekusi yang lebih buruk pada jumlah partisi yang kecil. Waktu eksekusi Spark pada partisi yang besar lebih cepat dibanding waktu eksekusi Hadoop terkecil. Aplikasi Spark lebih cepat dibanding Hadoop asalakan jumlah partisi diatur dengan benar.

Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang berbeda. Percobaan ini akan menggunakan 1 komputer sebagai komputer master dan 10 komputer lainnya sebagai worker dengan setiap worker menggunakan 1 core. Ukuran data yang digunakan adalah 2 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 30. Tabel (5.3) berikut adalah hasil dari eksperimen:

Tabel 5.3: Percobaan Jumlah Partisi Hadoop dan Spark dengan Ukuran Data 2 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark Tanpa standar Deviasi (Detik)	Waktu Eksekusi Spark (Detik)	Waktu Eksekusi Hadoop (Detik)	Hasil Reduksi Spark Tanpa standar Deviasi (GB)	Hasil Reduksi Spark (GB)	Hasil Reduksi Hadoop (GB)
2	10	929	958	365	0.96	1.2	1
2	30	350	317	355	0.96	1.2	1
2	50	205	217	428	0.96	1.2	1
2	100	126	134	446	0.96	1.2	1
2	150	110	109	489	0.96	1.2	1
2	200	98	99	545	0.96	1.2	1
2	250	83	86	589	0.96	1.2	1



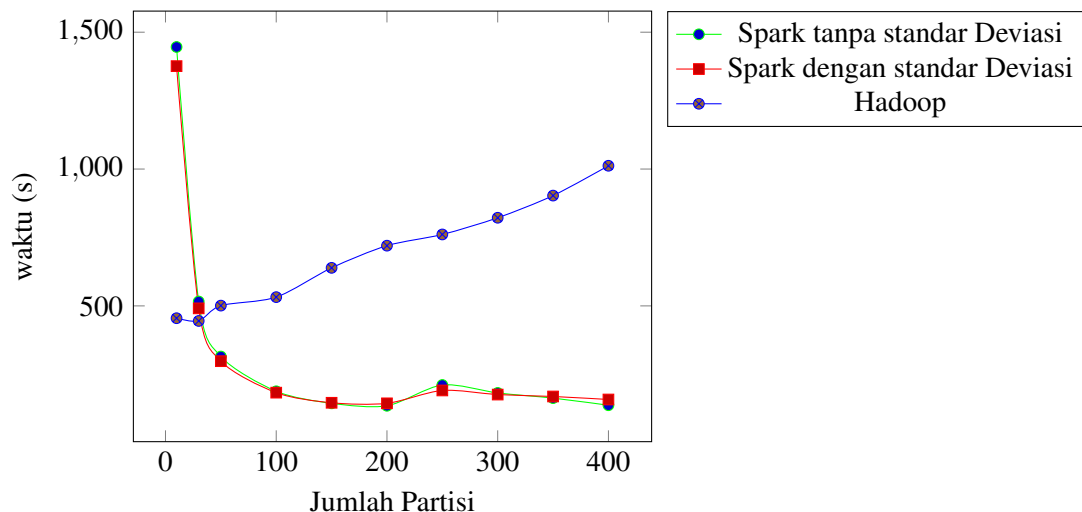
Gambar 5.7: Hasil Percobaan Partisi Spark dan Hadoop 2GB

Berdasarkan hasil grafik (5.7), dapat dilihat bahwa waktu eksekusi Spark menurun dan waktu eksekusi Hadoop meningkat ketika jumlah partisi diperbesar. Waktu eksekusi Hadoop menaik secara konsisten ketika jumlah partisi diperbesarkan. Waktu eksekusi Spark menurun drastis pada awalnya ketika jumlah partisi ditingkatkan sampai titik tertentu dimana peningkatan jumlah partisi tidak memiliki dampak yang sangat drastis pada waktu eksekusi Spark. Tidak ada perbedaan yang jauh antara waktu eksekusi aplikasi Spark dengan standar deviasi maupun yang tidak. Aplikasi Spark memiliki waktu eksekusi yang lebih baik dibanding Hadoop pada jumlah partisi yang besar dan waktu eksekusi yang lebih buruk pada jumlah partisi yang kecil. Waktu eksekusi Spark pada partisi yang besar lebih cepat dibanding waktu eksekusi Hadoop terkecil. Aplikasi Spark lebih cepat dibanding Hadoop asalakan jumlah partisi diatur dengan benar.

Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang berbeda. Percobaan ini akan menggunakan 1 komputer sebagai komputer master dan 10 komputer lainnya sebagai worker dengan setiap worker menggunakan 1 core. Ukuran data yang digunakan adalah 3 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 30. Tabel (5.4) berikut adalah hasil dari eksperimen:

Tabel 5.4: Percobaan Jumlah Partisi Hadoop dan Spark dengan Ukuran Data 3 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark Tanpa standar Deviasi (Detik)	Waktu Eksekusi Spark (Detik)	Waktu Eksekusi Hadoop (Detik)	Hasil Reduksi Spark Tanpa standar Deviasi (GB)	Hasil Reduksi Spark (GB)	Hasil Reduksi Hadoop (GB)
3	10	1446	1376	455	1.2	1.5	1.2
3	30	516	491	445	1.2	1.5	1.2
3	50	315	298	501	1.2	1.5	1.2
3	100	188	183	532	1.2	1.5	1.2
3	150	144	146	639	1.2	1.5	1.2
3	200	135	144	720	1.2	1.5	1.2
3	250	211	191	761	1.2	1.5	1.2
3	300	182	176	822	1.2	1.5	1.2
3	350	163	169	903	1.2	1.5	1.2
3	400	137	158	1012	1.2	1.5	1.2



Gambar 5.8: Hasil Percobaan Partisi Spark dan Hadoop 3GB

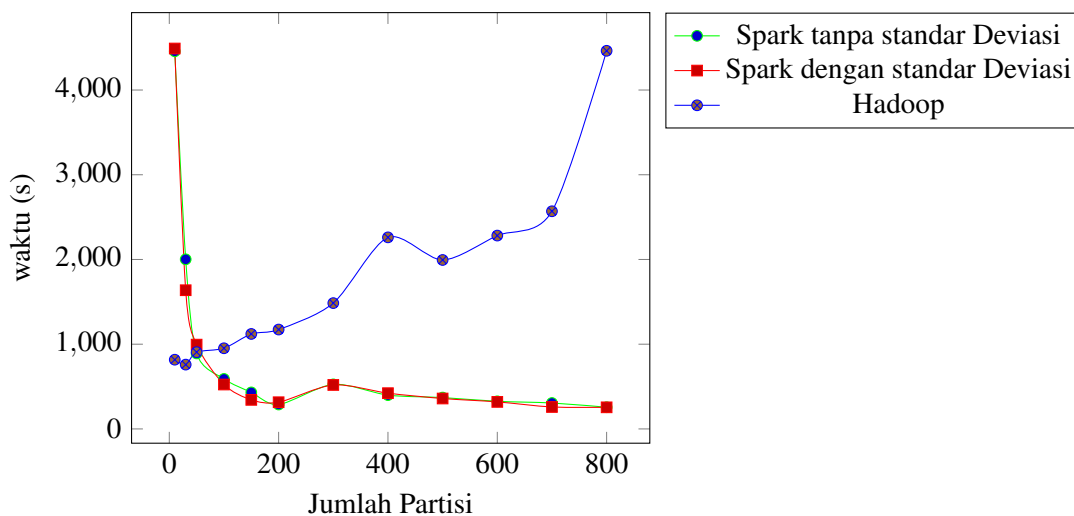
Berdasarkan hasil grafik (5.8), dapat dilihat bahwa waktu eksekusi Spark menurun dan waktu eksekusi Hadoop meningkat ketika jumlah partisi diperbesar. Waktu eksekusi Hadoop menaik secara konsisten ketika jumlah partisi diperbesarkan. Waktu eksekusi Spark menurun drastis pada awalnya ketika jumlah partisi ditingkatkan sampai titik tertentu dimana peningkatan jumlah partisi tidak memiliki dampak yang sangat drastis pada waktu eksekusi Spark. Tidak ada perbedaan yang jauh antara waktu eksekusi aplikasi Spark dengan standar deviasi maupun yang tidak. Aplikasi Spark memiliki waktu eksekusi yang lebih baik dibanding Hadoop pada jumlah partisi yang besar dan waktu eksekusi yang lebih buruk pada jumlah partisi yang kecil. Waktu eksekusi Spark pada partisi yang besar lebih cepat dibanding waktu eksekusi Hadoop terkecil. Aplikasi Spark lebih cepat dibanding Hadoop asalakan jumlah partisi diatur dengan benar.

Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang berbeda. Percobaan ini akan menggunakan 1 komputer sebagai komputer master dan 10 komputer lainnya sebagai worker dengan setiap worker menggunakan 1 core. Ukuran data yang digunakan adalah 5 GB.

- 1 Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah
- 2 objek maksimum untuk setiap *dendrogram* adalah 30. Tabel (5.5) berikut adalah hasil dari eksperimen:

Tabel 5.5: Percobaan Jumlah Partisi Hadoop dan Spark dengan Ukuran Data 5 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark Tanpa standar Deviasi (Detik)	Waktu Eksekusi Spark (Detik)	Waktu Eksekusi Hadoop (Detik)	Hasil Reduksi Spark Tanpa standar Deviasi (GB)	Hasil Reduksi Spark (GB)	Hasil Reduksi Hadoop (GB)
5	10	4490	4457	817	2.1	2.6	2.2
5	30	1637	2002	759	2.1	2.6	2.2
5	50	995	891	906	2.1	2.6	2.2
5	100	524	590	952	2.1	2.6	2.2
5	150	343	431	1121	2.1	2.6	2.2
5	200	315	288	1173	2.1	2.6	2.2
5	300	519	526	1485	2.1	2.6	2.2
5	400	422	399	2261	2.1	2.6	2.2
5	500	359	370	1994	2.1	2.6	2.2
5	600	319	326	2282	2.1	2.6	2.2
5	700	259	306	2569	2.1	2.6	2.2
5	800	255	256	4463	2.1	2.6	2.2



Gambar 5.9: Hasil Percobaan Partisi Spark dan Hadoop 5GB

4 Berdasarkan hasil grafik (5.9), dapat dilihat bahwa waktu eksekusi Spark menurun dan waktu eksekusi

5 Hadoop meningkat ketika jumlah partisi diperbesar. Waktu eksekusi Hadoop menaik secara konsisten

6 ketika jumlah partisi diperbesarkan. Waktu eksekusi Spark menurun drastis pada awalnya ketika jumlah

7 partisi ditingkatkan sampai titik tertentu dimana peningkatan jumlah partisi tidak memiliki dampak yang

8 sangat drastis pada waktu eksekusi Spark. Tidak ada perbedaan yang jauh antara waktu eksekusi aplikasi

9 Spark dengan standar deviasi maupun yang tidak. Aplikasi Spark memiliki waktu eksekusi yang lebih baik

10 dibanding Hadoop pada jumlah partisi yang besar dan waktu eksekusi yang lebih buruk pada jumlah partisi

11 yang kecil. Waktu eksekusi Spark pada partisi yang besar lebih cepat dibanding waktu eksekusi Hadoop

12 terkecil. Aplikasi Spark lebih cepat dibanding Hadoop asalakan jumlah partisi diatur dengan benar.

13

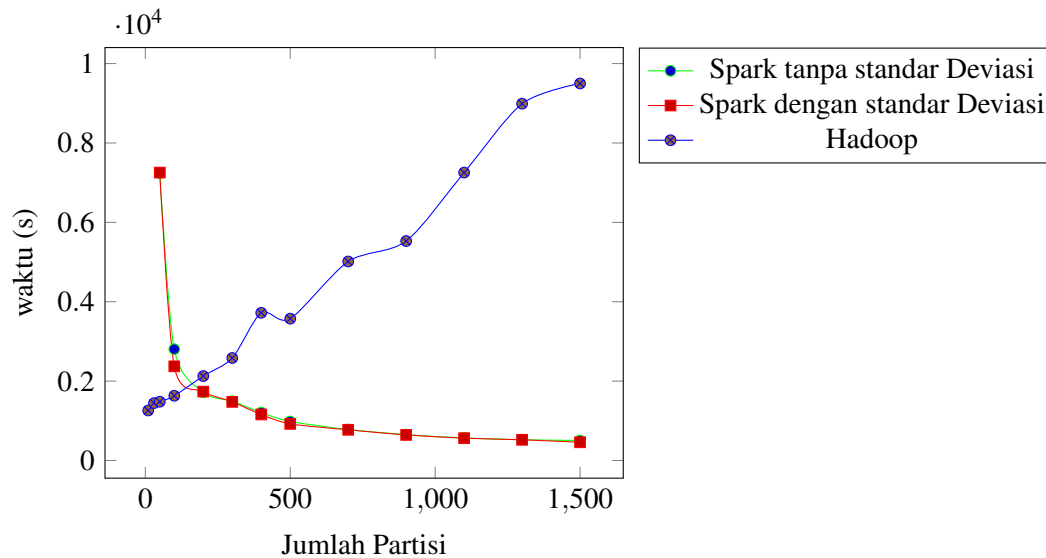
Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang berbeda. Percobaan ini akan menggunakan 1 komputer sebagai komputer master dan 10 komputer lainnya sebagai worker dengan setiap worker menggunakan 1 core. Ukuran data yang digunakan adalah 10 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 30. Tabel 5.6 dan Tabel 5.8 berikut adalah hasil dari eksperimen:

Tabel 5.6: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 10 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark Standar (Detik)	Waktu Eksekusi Spark Tanpa Deviasi (Detik)	Hasil Reduksi Spark Standar (GB)	Hasil Reduksi Spark Tanpa Deviasi (GB)
10	50	7254	7236	3.7	4.6
10	100	237	2805	3.7	4.6
10	200	1736	1718	3.7	4.6
10	300	1477	1494	3.7	4.6
10	400	1160	1207	3.7	4.6
10	500	923	984	3.7	4.6
10	600	774	780	3.7	4.6
10	700	645	652	3.7	4.6
10	900	563	568	3.7	4.6
10	1100	522	524	3.7	4.6
10	1300	359	504	3.7	4.6
10	1500	255	256	3.7	4.6

Tabel 5.7: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 10 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (Detik)	Hasil Reduksi Hadoop (GB)
10	10	1260	3.9
10	30	1446	3.9
10	50	1481	3.9
10	100	1631	3.9
10	200	2127	3.9
10	300	2583	3.9
10	400	3721	3.9
10	500	3573	3.9
10	700	5014	3.9
10	900	5529	3.9
10	1100	7254	3.9
10	1300	8989	3.9
10	1500	9499	3.9



Gambar 5.10: Hasil Percobaan Partisi Spark dan Hadoop 10GB

Berdasarkan hasil grafik (5.10), dapat dilihat bahwa waktu eksekusi Spark menurun dan waktu eksekusi Hadoop meningkat ketika jumlah partisi diperbesar. Waktu eksekusi Hadoop menaik secara konsisten ketika jumlah partisi diperbesarkan. Waktu eksekusi Spark menurun drastis pada awalnya ketika jumlah partisi ditingkatkan sampai titik tertentu dimana peningkatan jumlah partisi tidak memiliki dampak yang sangat drastis pada waktu eksekusi Spark. Tidak ada perbedaan yang jauh antara waktu eksekusi aplikasi Spark dengan standar deviasi maupun yang tidak. Aplikasi Spark memiliki waktu eksekusi yang lebih baik dibanding Hadoop pada jumlah partisi yang besar dan waktu eksekusi yang lebih buruk pada jumlah partisi yang kecil. Waktu eksekusi Spark pada partisi yang besar lebih cepat dibanding waktu eksekusi Hadoop terkecil. Aplikasi Spark lebih cepat dibanding Hadoop asalakan jumlah partisi diatur dengan benar.

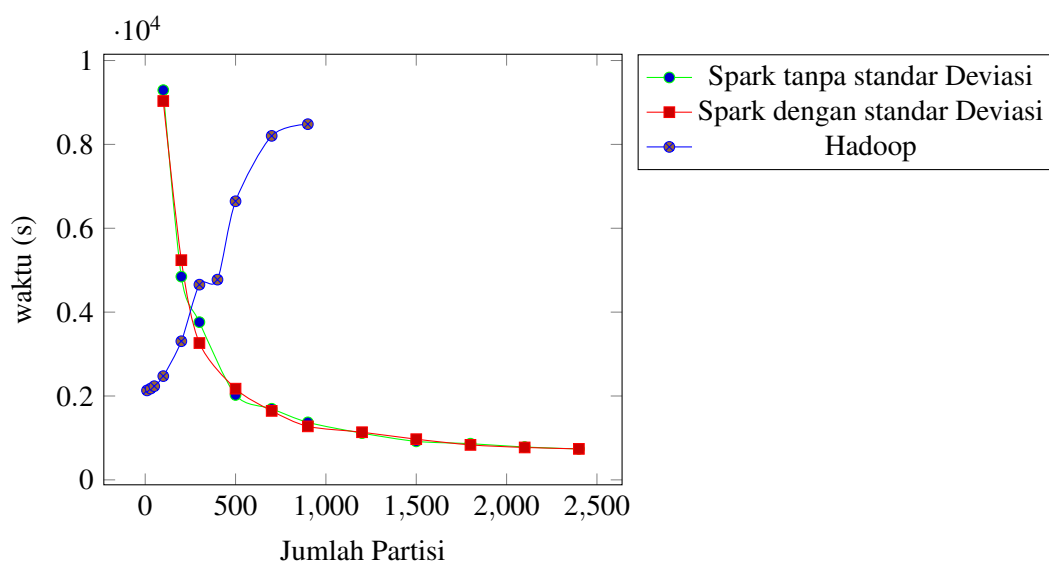
Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang berbeda. Percobaan ini akan menggunakan 1 komputer sebagai komputer master dan 10 komputer lainnya sebagai worker dengan setiap worker menggunakan 1 core. Ukuran data yang digunakan adalah 15 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 30. Tabel (5.8 dan Tabel (5.9) berikut adalah hasil dari eksperimen:

Tabel 5.8: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 15

Ukuran Data)	Jumlah Partisi	Waktu Eksekusi Spark standar (GB)	Eksekusi Tanpa Deviasi	Waktu Eksekusi Spark (detik)	Hasil Reduksi Spark standar Deviasi)	Hasil Reduksi Spark)
15	100	9034		9294	5.8	7.3
15	200	5239		4847	5.8	7.3
15	300	3263		3761	5.8	7.3
15	500	2175		2024	5.8	7.3
15	700	1645		1696	5.8	7.3
15	900	1276		1372	5.8	7.3
15	1200	1136		1114	5.8	7.3
15	1500	970		918	5.8	7.3
15	1800	834		863	5.8	7.3
15	2100	773		783	5.8	7.3
15	2400	739		738	5.8	7.3

Tabel 5.9: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 15 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (detik)	Hasil Reduksi Hadoop (GB)
15	10	2133	3.9
15	30	2177	3.9
15	50	2234	3.9
15	100	2474	3.9
15	200	3306	3.9
15	300	4655	3.9
15	400	4775	3.9
15	500	6644	3.9
15	700	8203	3.9
15	900	8482	3.9



Gambar 5.11: Hasil Percobaan Partisi Spark dan Hadoop 15GB

Berdasarkan hasil grafik (5.11), dapat dilihat bahwa waktu eksekusi Spark menurun dan waktu eksekusi Hadoop meningkat ketika jumlah partisi diperbesar. Waktu eksekusi Hadoop menaik secara konsisten ketika jumlah partisi diperbesarkan. Waktu eksekusi Spark menurun drastis pada awalnya ketika jumlah partisi ditingkatkan sampai titik tertentu dimana peningkatan jumlah partisi tidak memiliki dampak yang sangat drastis pada waktu eksekusi Spark. Tidak ada perbedaan yang jauh antara waktu eksekusi aplikasi Spark dengan standar deviasi maupun yang tidak. Aplikasi Spark memiliki waktu eksekusi yang lebih baik dibanding Hadoop pada jumlah partisi yang besar dan waktu eksekusi yang lebih buruk pada jumlah partisi yang kecil. Waktu eksekusi Spark pada partisi yang besar lebih cepat dibanding waktu eksekusi Hadoop terkecil. Aplikasi Spark lebih cepat dibanding Hadoop asalakan jumlah partisi diatur dengan benar.

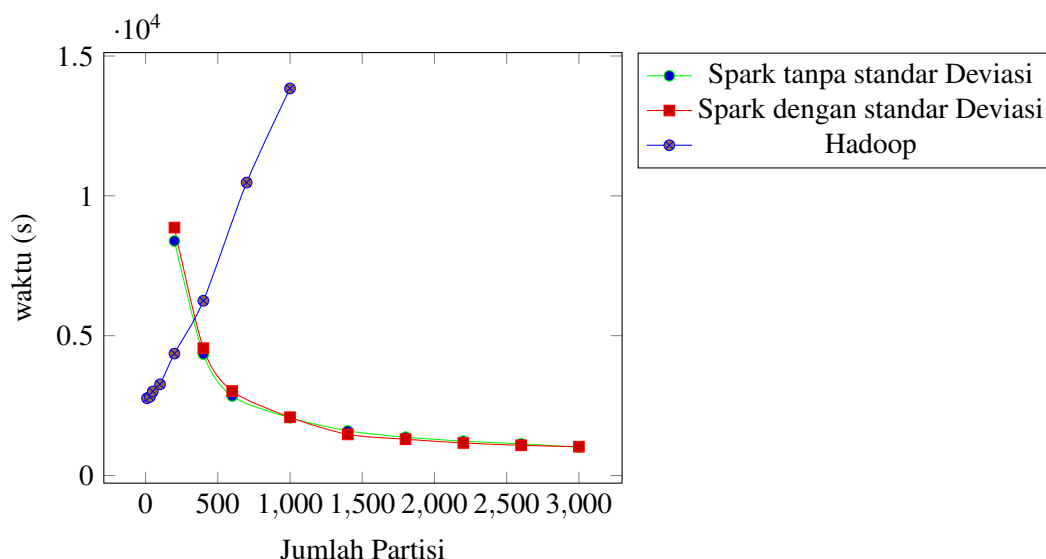
Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang berbeda. Percobaan ini akan menggunakan 1 komputer sebagai komputer master dan 10 komputer lainnya sebagai worker dengan setiap worker menggunakan 1 core. Ukuran data yang digunakan adalah 20 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 30. Tabel (5.10 dan Tabel (5.11) berikut adalah hasil dari eksperimen:

Tabel 5.10: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 20

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark Tanpa standar Deviasi (detik)	Waktu Eksekusi Spark (detik)	Hasil Reduksi Spark standar Tanpa Deviasi (GB)	Hasil Reduksi Spark (GB)
20	200	8866	8386	7.7	9.6
20	400	4553	4342	7.7	9.6
20	600	3021	2841	7.7	9.6
20	1000	2084	2065	7.7	9.6
20	1400	1471	1598	7.7	9.6
20	1800	1298	1372	7.7	9.6
20	2200	1165	1228	7.7	9.6
20	2600	1081	1133	7.7	9.6
20	3000	1031	1010	7.7	9.6

Tabel 5.11: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 20 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (detik)	Hasil Reduksi Hadoop (GB)
20	10	2763	8.1
20	30	2811	8.1
20	50	3007	8.1
20	100	3261	8.1
20	200	4360	8.1
20	400	6249	8.1
20	700	10476	8.1
20	1000	13839	8.1



Gambar 5.12: Hasil Percobaan Partisi Spark dan Hadoop 20GB

Berdasarkan hasil grafik (5.12), dapat dilihat bahwa waktu eksekusi Spark menurun dan waktu eksekusi Hadoop meningkat ketika jumlah partisi diperbesar. Waktu eksekusi Hadoop menaik secara konsisten ketika jumlah partisi diperbesarkan. Waktu eksekusi Spark menurun drastis pada awalnya ketika jumlah partisi ditingkatkan sampai titik tertentu dimana peningkatan jumlah partisi tidak memiliki dampak yang sangat drastis pada waktu eksekusi Spark. Tidak ada perbedaan yang jauh antara waktu eksekusi aplikasi Spark dengan standar deviasi maupun yang tidak. Aplikasi Spark memiliki waktu eksekusi yang lebih baik dibanding Hadoop pada jumlah partisi yang besar dan waktu eksekusi yang lebih buruk pada jumlah partisi yang kecil. Waktu eksekusi Spark pada partisi yang besar lebih cepat dibanding waktu eksekusi Hadoop terkecil. Aplikasi Spark lebih cepat dibanding Hadoop asalakan jumlah partisi diatur dengan benar.

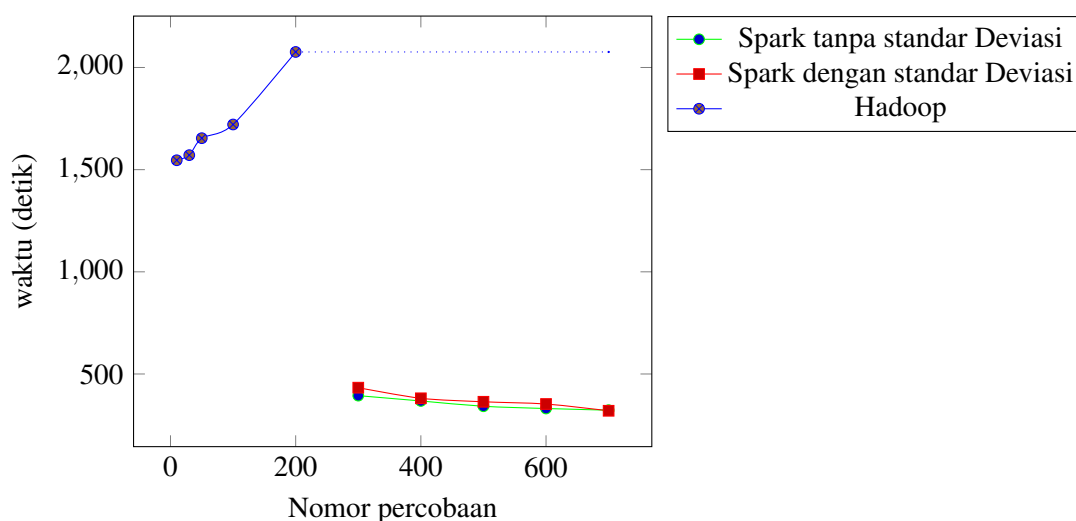
Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang optimal dengan jumlah objek maksimum adalah 50 untuk setiap *dendrogram*. Percobaan ini akan menggunakan 1 komputer sebagai komputer *master* dan 10 komputer lainnya sebagai *worker* dengan setiap *worker* menggunakan 1 core. Ukuran data yang digunakan adalah 5 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 50. Tabel (5.12) dan Tabel (5.13) berikut adalah hasil dari eksperimen:

Tabel 5.12: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 5 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (detik)	Hasil Reduksi Hadoop (GB)
5	10	1546	1.5
5	30	1571	1.5
5	50	1654	1.5
5	100	1721	1.5
5	200	2076	1.5

Tabel 5.13: Percobaan Jumlah Partisi Spark dengan Ukuran Data 5 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark standar (detik)	Waktu Eksekusi Spark Tanpa Deviasi	Hasil Spark standar (GB)	Hasil Spark Tanpa Deviasi	Reduksi Tanpa Deviasi	Reduksi Spark (GB)
5	300	394	433	1.5	1.9		
5	400	368	381	1.5	1.9		
5	500	342	364	1.5	1.9		
5	600	331	353	1.5	1.9		
5	700	323	320	1.5	1.9		



Gambar 5.13: Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 5 GB

Berdasarkan hasil grafik (5.13), dapat dilihat waktu Hadoop terus meningkat seiring meningkatnya jumlah partisi. Jumlah partisi yang dicoba pada Hadoop hanya mencapai 200 karena waktu eksekusi yang sudah berbeda jauh dibanding Spark dan untuk partisi yang lebih besar pasti diatas garis biru titik-titik sesuai pola peningkatan waktu eksekusi Hadoop terhadap jumlah partisi. Garis biru titik-titik merupakan waktu eksekusi Hadoop dengan jumlah partisi 200. Spark memiliki waktu eksekusi yang jauh lebih cepat dibanding Hadoop.

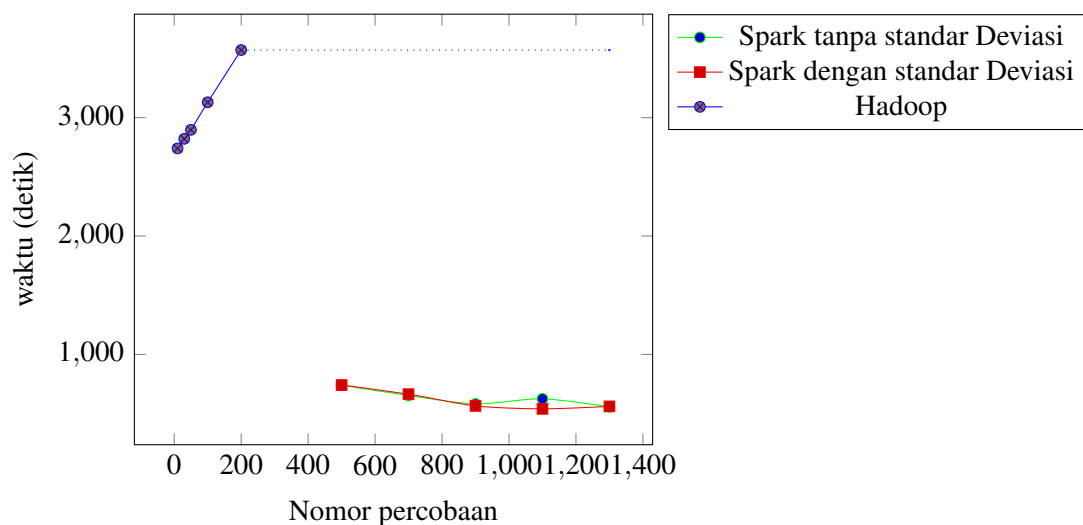
Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang optimal dengan jumlah objek maksimum adalah 50 untuk setiap *dendrogram*. Percobaan ini akan menggunakan 1 komputer sebagai komputer *master* dan 10 komputer lainnya sebagai *worker* dengan setiap *worker* menggunakan 1 core. Ukuran data yang digunakan adalah 10 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 50. Tabel (5.14) dan Tabel (5.15) berikut adalah hasil dari eksperimen:

Tabel 5.14: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 10 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (detik)	Hasil Reduksi Hadoop (GB)
10	10	2740	2.7
10	30	2821	2.7
10	50	2897	2.7
10	100	3130	2.7
10	200	3571	2.7

Tabel 5.15: Percobaan Jumlah Partisi Spark dengan Ukuran Data 10 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark standar (detik)	Eksekusi Tanpa Deviasi	Waktu Eksekusi Spark (detik)	Hasil Reduksi Spark standar (GB)	Hasil Reduksi Spark (GB)
10	500	740		741	2.7	3.3
10	700	653		664	2.7	3.3
10	900	582		565	2.7	3.3
10	1100	625		540	2.7	3.3
10	1300	557		561	2.7	3.3



Gambar 5.14: Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 10 GB

Berdasarkan hasil grafik (5.14), dapat dilihat waktu Hadoop terus meningkat seiring meningkatnya jumlah partisi. Jumlah partisi yang dicoba pada Hadoop hanya mencapai 200 karena waktu eksekusi yang sudah berbeda jauh dibanding Spark dan untuk partisi yang lebih besar pasti diatas garis biru titik-titik sesuai pola peningkatan waktu eksekusi Hadoop terhadap jumlah partisi. Garis biru titik-titik merupakan waktu eksekusi Hadoop dengan jumlah partisi 200. Spark memiliki waktu eksekusi yang jauh lebih cepat dibanding Hadoop.

Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang optimal dengan jumlah objek maksimum adalah 50 untuk setiap *dendrogram*. Percobaan ini akan menggunakan 1 komputer sebagai komputer *master* dan 10 komputer lainnya sebagai *worker* dengan setiap *worker*

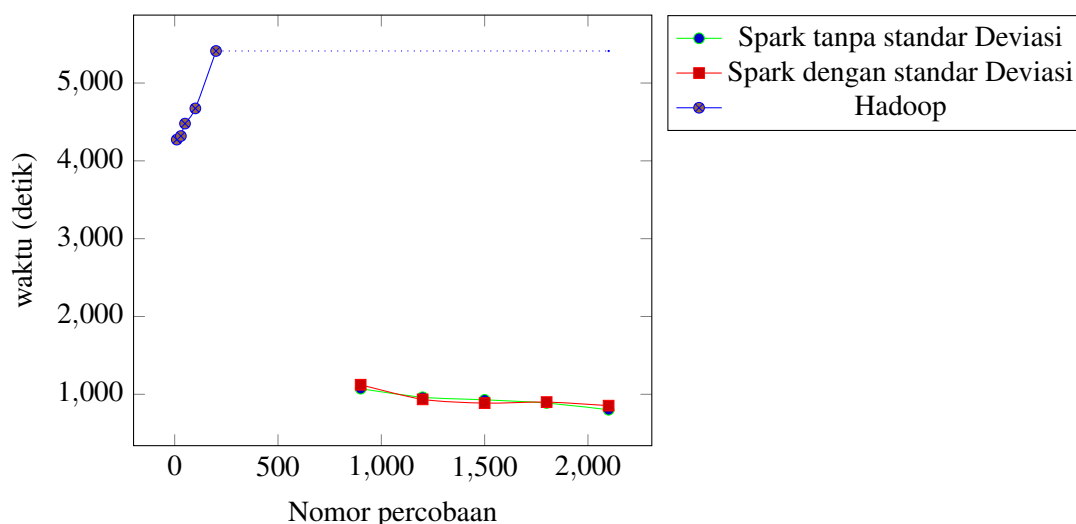
- 1 menggunakan 1 core. Ukuran data yang digunakan adalah 15 GB. Metode yang digunakan adalah metode
 2 *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram*
 3 adalah 50. Tabel (5.16) dan Tabel (5.17) berikut adalah hasil dari eksperimen:

Tabel 5.16: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 15 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (detik)	Hasil Reduksi Hadoop (GB)
15	10	4273	4.2
15	30	4319	4.2
15	50	4479	4.2
15	100	4674	4.2
15	200	5412	4.2

Tabel 5.17: Percobaan Jumlah Partisi Spark dengan Ukuran Data 15 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark standar (detik)	Eksekusi Tanpa Deviasi	Waktu Eksekusi Spark (detik)	Hasil Reduksi Spark standar (GB)	Hasil Reduksi Tanpa Deviasi	Hasil Reduksi Spark (GB)
15	900	1072		1123	4.2		5.2
15	1200	962		935	4.2		5.2
15	1500	929		887	4.2		5.2
15	1800	888		900	4.2		5.2
15	2100	801		854	4.2		5.2



Gambar 5.15: Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 15 GB

- 5 Berdasarkan hasil grafik (5.15), dapat dilihat waktu Hadoop terus meningkat seiring meningkatnya
 6 jumlah partisi. Jumlah partisi yang dicoba pada Hadoop hanya mencapai 200 karena waktu eksekusi yang
 7 sudah berbeda jauh dibanding Spark dan untuk partisi yang lebih besar pasti diatas garis biru titik-titik sesuai
 8 pola peningkatan waktu eksekusi Hadoop terhadap jumlah partisi. Garis biru titik-titik merupakan waktu
 9 eksekusi Hadoop dengan jumlah partisi 200. Spark memiliki waktu eksekusi yang jauh lebih cepat dibanding
 10 Hadoop.
 11

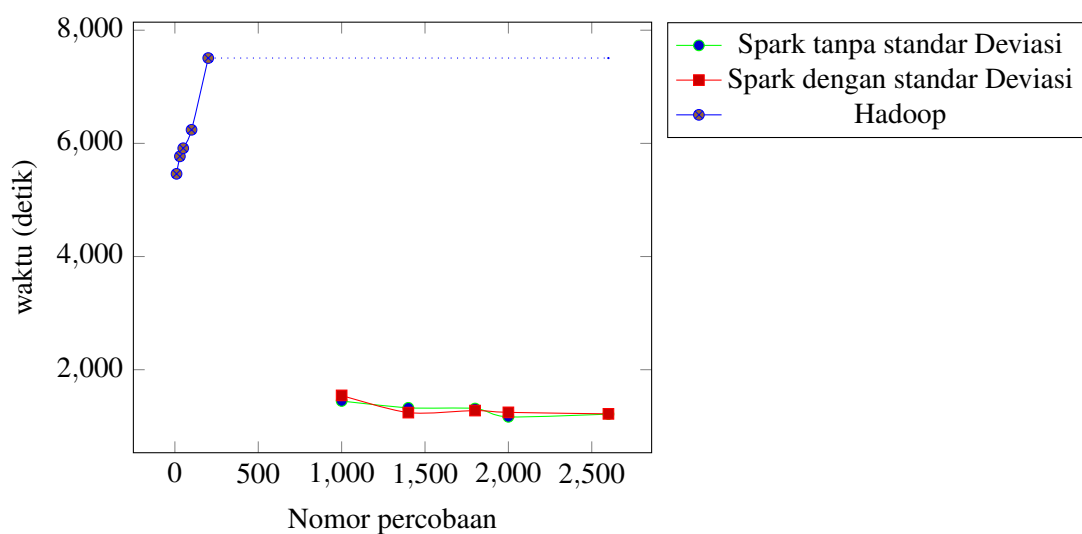
Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang optimal dengan jumlah objek maksimum adalah 50 untuk setiap *dendrogram*. Percobaan ini akan menggunakan 1 komputer sebagai komputer *master* dan 10 komputer lainnya sebagai *worker* dengan setiap *worker* menggunakan 1 core. Ukuran data yang digunakan adalah 20 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 50. Tabel (5.18) dan Tabel (5.19) berikut adalah hasil dari eksperimen:

Tabel 5.18: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 20 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (detik)	Hasil Reduksi Hadoop (GB)
20	10	5462	5.6
20	30	5771	5.6
20	50	5914	5.6
20	100	6240	5.6
20	200	7508	5.6

Tabel 5.19: Percobaan Jumlah Partisi Spark dengan Ukuran Data 20 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark Tanpa standar Deviasi (detik)	Waktu Eksekusi Spark (detik)	Hasil Reduksi Spark standar (GB)	Hasil Reduksi Spark Tanpa standar Deviasi (GB)
20	1000	1447	1546	5.6	6.9
20	1400	1327	1242	5.6	6.9
20	1800	1314	1278	5.6	6.9
20	2200	1167	1246	5.6	6.9
20	2600	1216	1220	5.6	6.9



Gambar 5.16: Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 20 GB

Berdasarkan hasil grafik (5.16), dapat dilihat waktu Hadoop terus meningkat seiring meningkatnya jumlah partisi. Jumlah partisi yang dicoba pada Hadoop hanya mencapai 200 karena waktu eksekusi yang

sudah berbeda jauh dibanding Spark dan untuk partisi yang lebih besar pasti diatas garis biru titik-titik sesuai pola peningkatan waktu eksekusi Hadoop terhadap jumlah partisi. Garis biru titik-titik merupakan waktu eksekusi Hadoop dengan jumlah partisi 200. Spark memiliki waktu eksekusi yang jauh lebih cepat dibanding Hadoop.

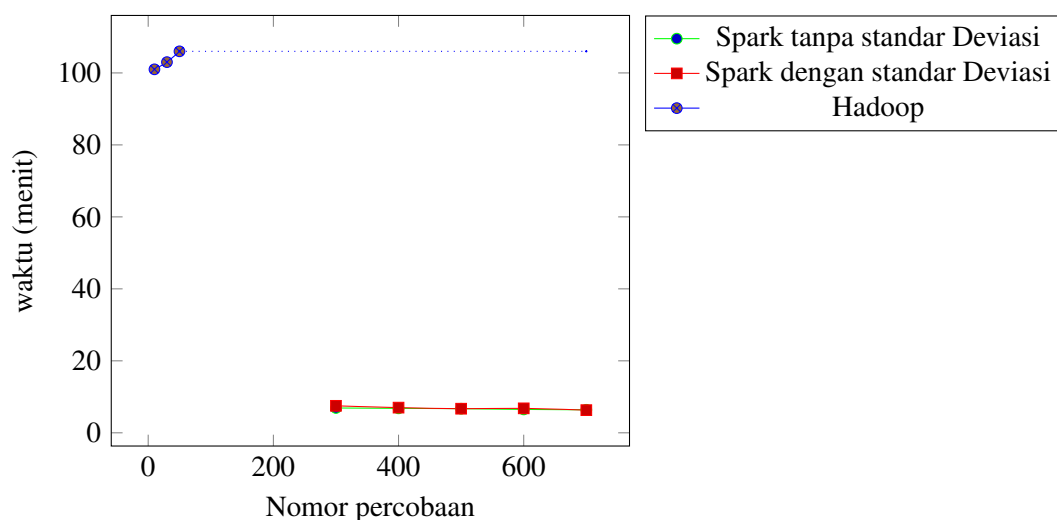
Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang optimal dengan jumlah objek maksimum adalah 100 untuk setiap *dendrogram*. Percobaan ini akan menggunakan 1 komputer sebagai komputer *master* dan 10 komputer lainnya sebagai *worker* dengan setiap *worker* menggunakan 1 core. Ukuran data yang digunakan adalah 5 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 100. Tabel (5.20) dan Tabel (5.21) berikut adalah hasil dari eksperimen:

Tabel 5.20: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 5 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (menit)	Hasil Reduksi Hadoop (GB)
5	10	101	0.962
5	30	103	0.962
5	50	106	0.962

Tabel 5.21: Percobaan Jumlah Partisi Spark dengan Ukuran Data 5 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark standar (menit)	Waktu Eksekusi Spark Tanpa Deviasi (menit)	Hasil Reduksi Spark standar (GB)	Hasil Reduksi Spark Tanpa Deviasi (GB)
5	300	6.9	7.5	1	1.2
5	400	6.8	7.0	1	1.2
5	500	6.7	6.7	1	1.2
5	600	6.5	6.8	1	1.2
5	700	6.4	6.3	1	1.2



Gambar 5.17: Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 5 GB

Berdasarkan hasil grafik (5.17), dapat dilihat waktu Hadoop terus meningkat seiring meningkatnya jumlah partisi. Jumlah partisi yang dicoba pada Hadoop hanya mencapai 200 karena waktu eksekusi yang sudah berbeda jauh dibanding Spark dan untuk partisi yang lebih besar pasti diatas garis biru titik-titik sesuai pola peningkatan waktu eksekusi Hadoop terhadap jumlah partisi. Garis biru titik-titik merupakan waktu eksekusi Hadoop dengan jumlah partisi 200. Spark memiliki waktu eksekusi yang jauh lebih cepat dibanding Hadoop.

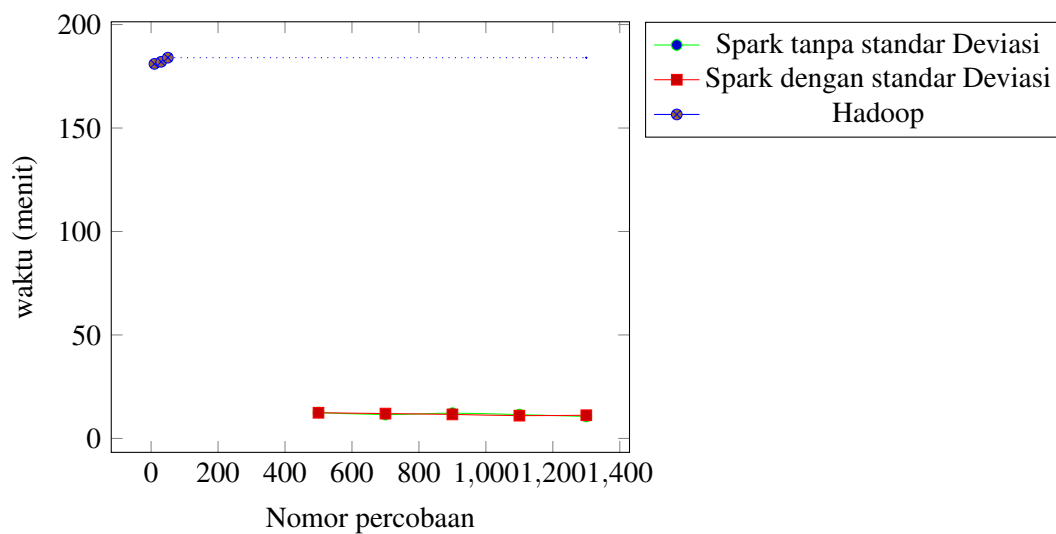
Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang optimal dengan jumlah objek maksimum adalah 100 untuk setiap *dendrogram*. Percobaan ini akan menggunakan 1 komputer sebagai komputer *master* dan 10 komputer lainnya sebagai *worker* dengan setiap *worker* menggunakan 1 core. Ukuran data yang digunakan adalah 10 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 100. Tabel (5.22) dan Tabel (5.23) berikut adalah hasil dari eksperimen:

Tabel 5.22: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 10 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (menit)	Hasil Reduksi Hadoop (GB)
10	10	181	1.7
10	30	182	1.7
10	50	184	1.7

Tabel 5.23: Percobaan Jumlah Partisi Spark dengan Ukuran Data 10 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark Tanpa standar Deviasi (menit)	Waktu Eksekusi Spark (menit)	Hasil Reduksi Spark standar (GB)	Hasil Reduksi Spark Tanpa Deviasi (GB)
10	500	12.5	12.4	1.8	2.2
10	700	11.5	12.0	1.8	2.2
10	900	12.2	11.6	1.8	2.2
10	1100	11.5	11.0	1.8	2.2
10	1300	10.6	11.2	1.8	2.2



Gambar 5.18: Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 10 GB

Berdasarkan hasil grafik (5.18), dapat dilihat waktu Hadoop terus meningkat seiring meningkatnya jumlah partisi. Jumlah partisi yang dicoba pada Hadoop hanya mencapai 200 karena waktu eksekusi yang sudah berbeda jauh dibanding Spark dan untuk partisi yang lebih besar pasti diatas garis biru titik-titik sesuai pola peningkatan waktu eksekusi Hadoop terhadap jumlah partisi. Garis biru titik-titik merupakan waktu eksekusi Hadoop dengan jumlah partisi 200. Spark memiliki waktu eksekusi yang jauh lebih cepat dibanding Hadoop.

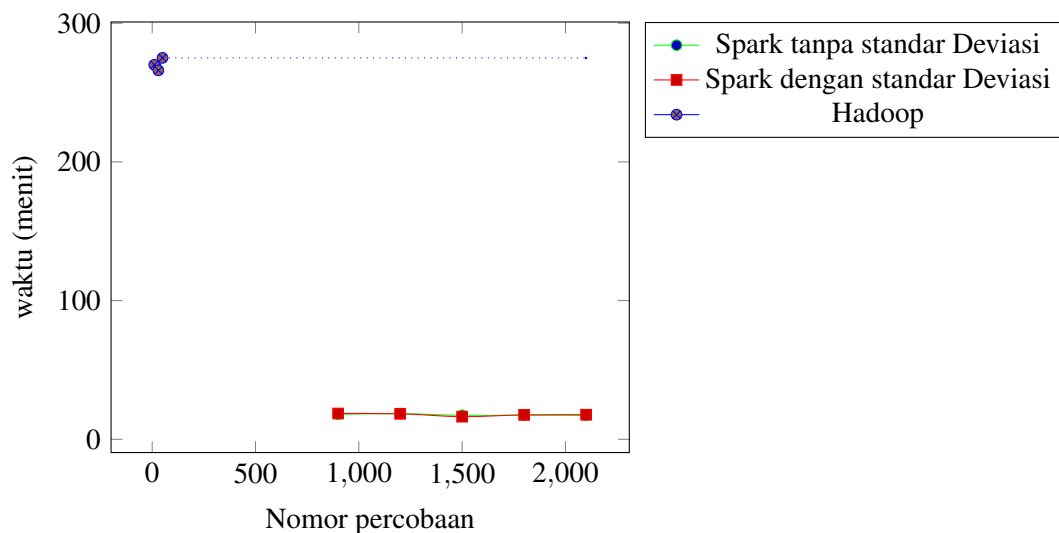
Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang optimal dengan jumlah objek maksimum adalah 100 untuk setiap *dendrogram*. Percobaan ini akan menggunakan 1 komputer sebagai komputer *master* dan 10 komputer lainnya sebagai *worker* dengan setiap *worker* menggunakan 1 core. Ukuran data yang digunakan adalah 15 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 100. Tabel (5.24) dan Tabel (5.25) berikut adalah hasil dari eksperimen:

Tabel 5.24: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 15 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (menit)	Hasil Reduksi Hadoop (GB)
15	10	270	2.6
15	30	266	2.6
15	50	275	2.6

Tabel 5.25: Percobaan Jumlah Partisi Spark dengan Ukuran Data 15 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark standar (menit)	Waktu Eksekusi Spark Tanpa Deviasi	Hasil Spark standar (GB)	Hasil Spark Reduksi Tanpa Deviasi
15	900	18.0	18.6	2.8	3.4
15	1200	18.5	18.4	2.8	3.4
15	1500	17.3	16.3	2.8	3.4
15	1800	17.4	17.6	2.8	3.4
15	2100	17.3	17.7	2.8	3.4



Gambar 5.19: Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 15 GB

Berdasarkan hasil grafik (5.19), dapat dilihat waktu Hadoop terus meningkat seiring meningkatnya jumlah partisi. Jumlah partisi yang dicoba pada Hadoop hanya mencapai 200 karena waktu eksekusi yang sudah berbeda jauh dibanding Spark dan untuk partisi yang lebih besar pasti diatas garis biru titik-titik sesuai pola peningkatan waktu eksekusi Hadoop terhadap jumlah partisi. Garis biru titik-titik merupakan waktu eksekusi Hadoop dengan jumlah partisi 200. Spark memiliki waktu eksekusi yang jauh lebih cepat dibanding Hadoop.

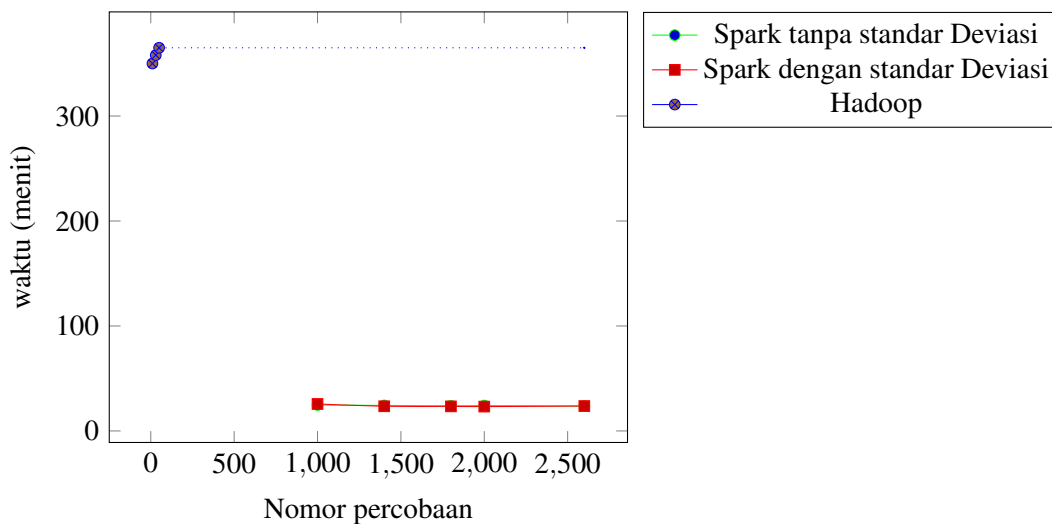
Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang optimal dengan jumlah objek maksimum adalah 100 untuk setiap *dendrogram*. Percobaan ini akan menggunakan 1 komputer sebagai komputer *master* dan 10 komputer lainnya sebagai *worker* dengan setiap *worker* menggunakan 1 core. Ukuran data yang digunakan adalah 20 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 100. Tabel (5.26) dan Tabel (5.27) berikut adalah hasil dari eksperimen:

Tabel 5.26: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 20 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (menit)	Hasil Reduksi Hadoop (GB)
20	10	350	3.5
20	30	358	3.5
20	50	365	3.5

Tabel 5.27: Percobaan Jumlah Partisi Spark dengan Ukuran Data 20 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark standar (menit)	Waktu Eksekusi Spark Tanpa Deviasi (menit)	Hasil Reduksi Spark standar (GB)	Hasil Reduksi Spark Tanpa Deviasi (GB)
20	1000	24.7	25.7	3.7	4.5
20	1400	24.3	23.5	3.7	4.5
20	1800	24.0	23.4	3.7	4.5
20	2200	24.2	23.2	3.7	4.5
20	2600	23.8	23.7	3.7	4.5



Gambar 5.20: Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 20 GB

Berdasarkan hasil grafik (5.20), dapat dilihat waktu Hadoop terus meningkat seiring meningkatnya jumlah partisi. Jumlah partisi yang dicoba pada Hadoop hanya mencapai 200 karena waktu eksekusi yang sudah berbeda jauh dibanding Spark dan untuk partisi yang lebih besar pasti diatas garis biru titik-titik sesuai pola peningkatan waktu eksekusi Hadoop terhadap jumlah partisi. Garis biru titik-titik merupakan waktu eksekusi Hadoop dengan jumlah partisi 200. Spark memiliki waktu eksekusi yang jauh lebih cepat dibanding Hadoop.

Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang optimal dengan jumlah objek maksimum adalah 30 untuk setiap *dendrogram*. Satu komputer akan digunakan sebagai komputer *master* dan 10 komputer lainnya sebagai *worker* dengan setiap *worker* menggunakan 3 core. Ukuran data yang digunakan adalah 5 GB. Metode yang digunakan adalah metode *single linkage*, dengan

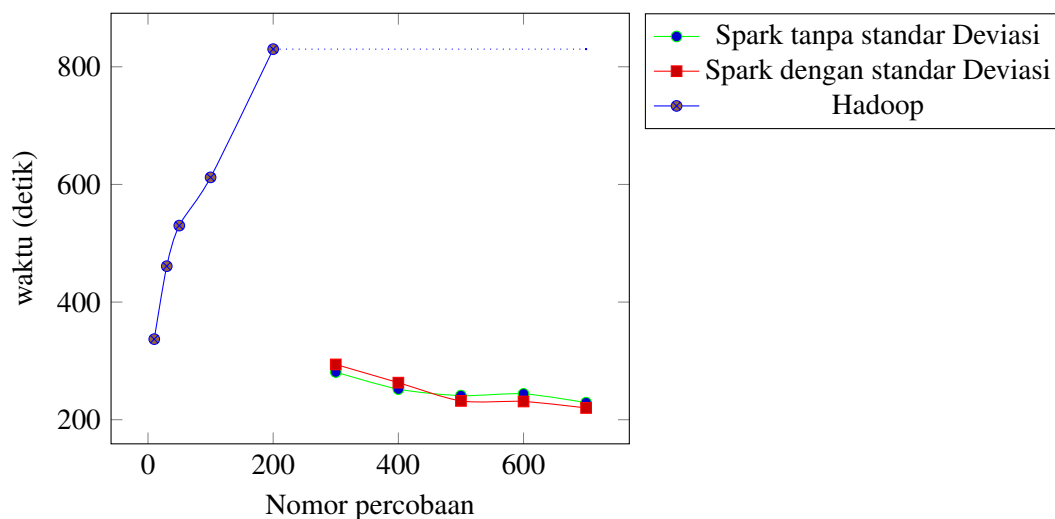
- 1 nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 30. Tabel
- 2 (5.36) dan Tabel (5.29) berikut adalah hasil dari eksperimen:

Tabel 5.28: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 5 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (detik)	Hasil Reduksi Hadoop (GB)
5	10	337	2.2
5	30	461	2.2
5	50	530	2.2
5	100	612	2.2
5	200	830	2.2

Tabel 5.29: Percobaan Jumlah Partisi Spark dengan Ukuran Data 5 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark standar (detik)	Waktu Eksekusi Spark Tanpa Deviasi (detik)	Hasil Reduksi Spark standar (GB)	Hasil Reduksi Spark Tanpa Deviasi (GB)
5	300	281	294	2.1	2.6
5	400	252	263	2.1	2.6
5	500	241	232	2.1	2.6
5	600	244	231	2.1	2.6
5	700	229	220	2.1	2.6



Gambar 5.21: Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 5 GB

Berdasarkan hasil grafik (5.21), dapat dilihat waktu Hadoop terus meningkat seiring meningkatnya jumlah partisi. Jumlah partisi yang dicoba pada Hadoop hanya mencapai 200 karena waktu eksekusi yang sudah berbeda jauh dibanding Spark dan untuk partisi yang lebih besar pasti diatas garis biru titik-titik sesuai pola peningkatan waktu eksekusi Hadoop terhadap jumlah partisi. Garis biru titik-titik merupakan waktu eksekusi Hadoop dengan jumlah partisi 200. Spark memiliki waktu eksekusi yang jauh lebih cepat dibanding Hadoop.

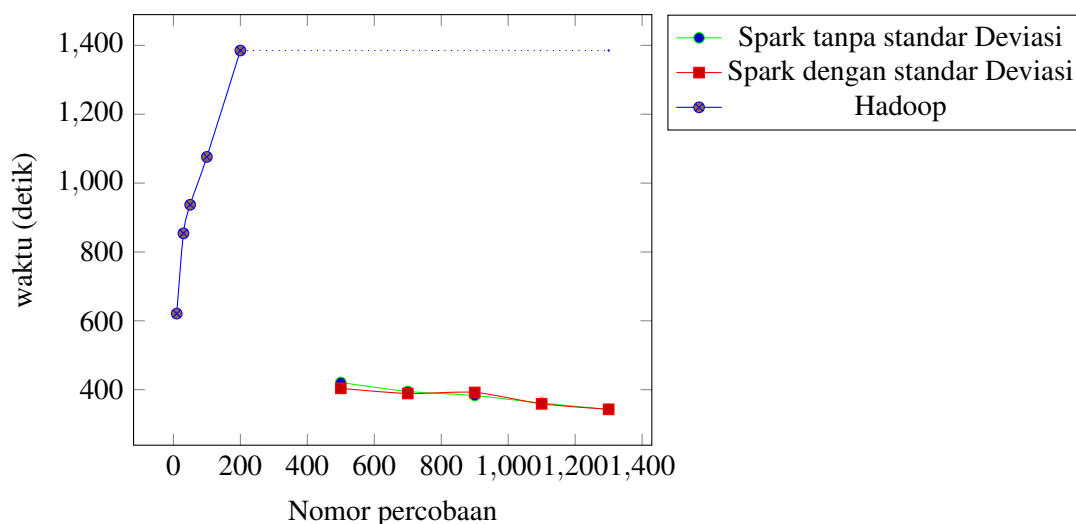
Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang optimal dengan jumlah objek maksimum adalah 30 untuk setiap *dendrogram*. Satu komputer akan digunakan sebagai komputer *master* dan 10 komputer lainnya sebagai *worker* dengan setiap *worker* menggunakan 3 core. Ukuran data yang digunakan adalah 10 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 30. Tabel (5.30) dan Tabel (??) berikut adalah hasil dari eksperimen:

Tabel 5.30: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 10 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (detik)	Hasil Reduksi Hadoop (GB)
10	10	621	3.9
10	30	854	3.9
10	50	937	3.9
10	100	1076	3.9
10	200	1385	3.9

Tabel 5.31: Percobaan Jumlah Partisi Spark dengan Ukuran Data 10 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark standar (detik)	Waktu Eksekusi Spark Tanpa Deviasi (detik)	Hasil Reduksi Spark standar (GB)	Hasil Reduksi Spark Tanpa Deviasi (GB)
10	500	421	404	4.6	3.7
10	700	395	389	4.6	3.7
10	900	383	392	4.6	3.7
10	1100	361	359	4.6	3.7
10	1300	343	343	4.6	3.7



Gambar 5.22: Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 10 GB

Berdasarkan hasil grafik (5.22), dapat dilihat waktu Hadoop terus meningkat seiring meningkatnya jumlah partisi. Jumlah partisi yang dicoba pada Hadoop hanya mencapai 200 karena waktu eksekusi yang sudah berbeda jauh dibanding Spark dan untuk partisi yang lebih besar pasti diatas garis biru titik-titik sesuai

1 pola peningkatan waktu eksekusi Hadoop terhadap jumlah partisi. Garis biru titik-titik merupakan waktu
 2 eksekusi Hadoop dengan jumlah partisi 200. Spark memiliki waktu eksekusi yang jauh lebih cepat dibanding
 3 Hadoop.

4

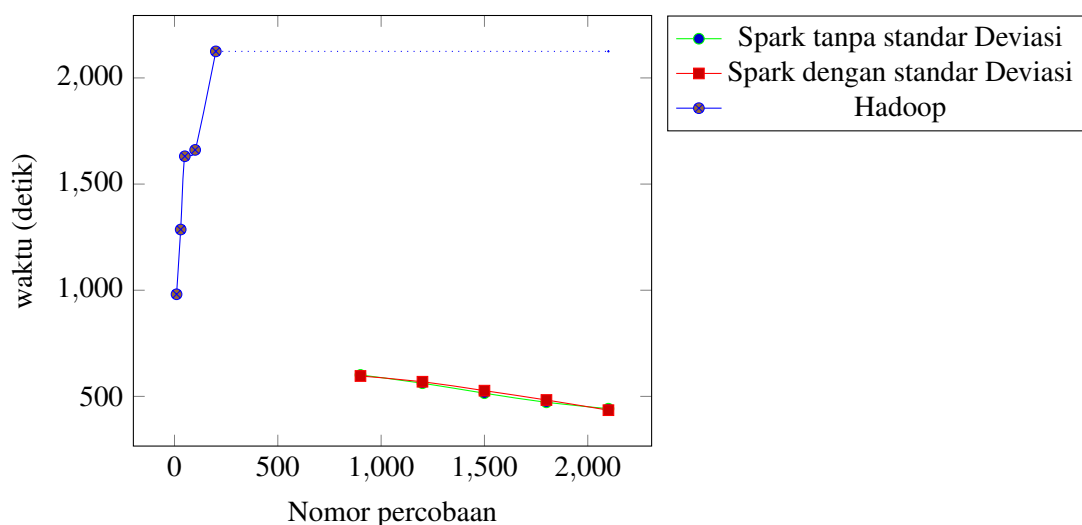
5 Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang
 6 optimal dengan jumlah objek maksimum adalah 30 untuk setiap *dendrogram*. Satu komputer akan digunakan
 7 sebagai komputer *master* dan 10 komputer lainnya sebagai *worker* dengan setiap *worker* menggunakan 3 core.
 8 Ukuran data yang digunakan adalah 15 GB. Metode yang digunakan adalah metode *single linkage*, dengan
 9 nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 30. Tabel
 10 (5.24) dan Tabel (5.25) berikut adalah hasil dari eksperimen:

Tabel 5.32: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 15 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (detik)	Hasil Reduksi Hadoop (GB)
15	10	981	6.1
15	30	1286	6.1
15	50	1631	6.1
15	100	1661	6.1
15	200	2125	6.1

Tabel 5.33: Percobaan Jumlah Partisi Spark dengan Ukuran Data 15 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark standar (detik)	Waktu Eksekusi Spark Tanpa Deviasi (detik)	Hasil Reduksi Spark standar (GB)	Hasil Reduksi Spark Tanpa Deviasi (GB)
15	900	601	596	5.8	7.3
15	1200	562	569	5.8	7.3
15	1500	515	527	5.8	7.3
15	1800	472	483	5.8	7.3
15	2100	442	435	5.8	7.3



Gambar 5.23: Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 15 GB

Berdasarkan hasil grafik (5.23), dapat dilihat waktu Hadoop terus meningkat seiring meningkatnya jumlah partisi. Jumlah partisi yang dicoba pada Hadoop hanya mencapai 200 karena waktu eksekusi yang sudah berbeda jauh dibanding Spark dan untuk partisi yang lebih besar pasti diatas garis biru titik-titik sesuai pola peningkatan waktu eksekusi Hadoop terhadap jumlah partisi. Garis biru titik-titik merupakan waktu eksekusi Hadoop dengan jumlah partisi 200. Spark memiliki waktu eksekusi yang jauh lebih cepat dibanding Hadoop.

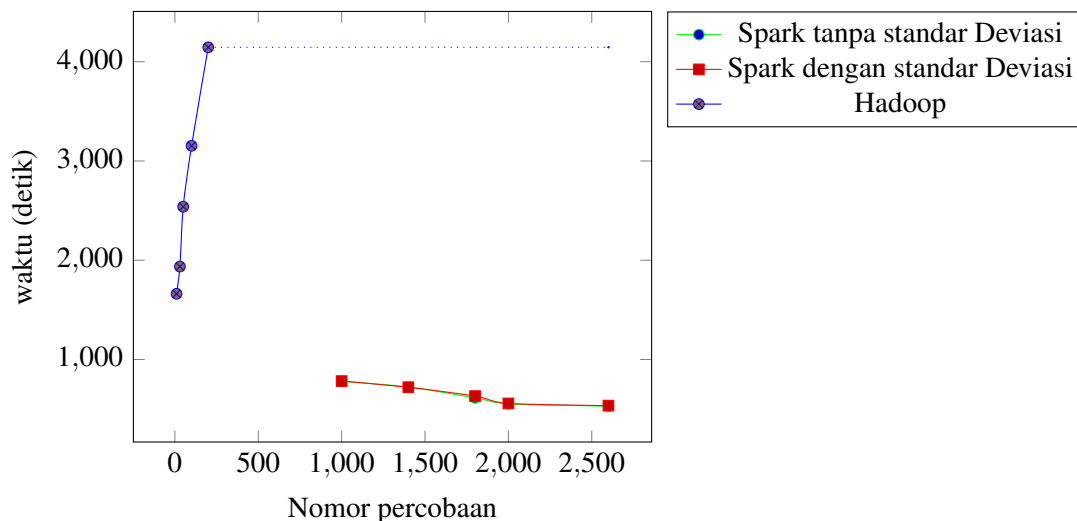
Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang optimal dengan jumlah objek maksimum adalah 30 untuk setiap *dendrogram*. Satu komputer akan digunakan sebagai komputer *master* dan 10 komputer lainnya sebagai *worker* dengan setiap *worker* menggunakan 3 core. Ukuran data yang digunakan adalah 20 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 30. Tabel (5.34) dan Tabel (5.35) berikut adalah hasil dari eksperimen:

Tabel 5.34: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 20 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (detik)	Hasil Reduksi Hadoop (GB)
20	10	1662	8.1
20	30	1936	8.1
20	50	2539	8.1
20	100	3153	8.1
20	200	4145	8.1

Tabel 5.35: Percobaan Jumlah Partisi Spark dengan Ukuran Data 20 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark Tanpa standar Deviasi (detik)	Waktu Eksekusi Spark (detik)	Hasil Reduksi Spark standar Tanpa Deviasi (GB)	Hasil Reduksi Spark (GB)
20	1000	780	782	7.7	9.6
20	1400	723	720	7.7	9.6
20	1800	613	631	7.7	9.6
20	2200	553	557	7.7	9.6
20	2600	530	535	7.7	9.6



Gambar 5.24: Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 20 GB

Berdasarkan hasil grafik (5.24), dapat dilihat waktu Hadoop terus meningkat seiring meningkatnya jumlah partisi. Jumlah partisi yang dicoba pada Hadoop hanya mencapai 200 karena waktu eksekusi yang sudah berbeda jauh dibanding Spark dan untuk partisi yang lebih besar pasti diatas garis biru titik-titik sesuai pola peningkatan waktu eksekusi Hadoop terhadap jumlah partisi. Garis biru titik-titik merupakan waktu eksekusi Hadoop dengan jumlah partisi 200. Spark memiliki waktu eksekusi yang jauh lebih cepat dibanding Hadoop.

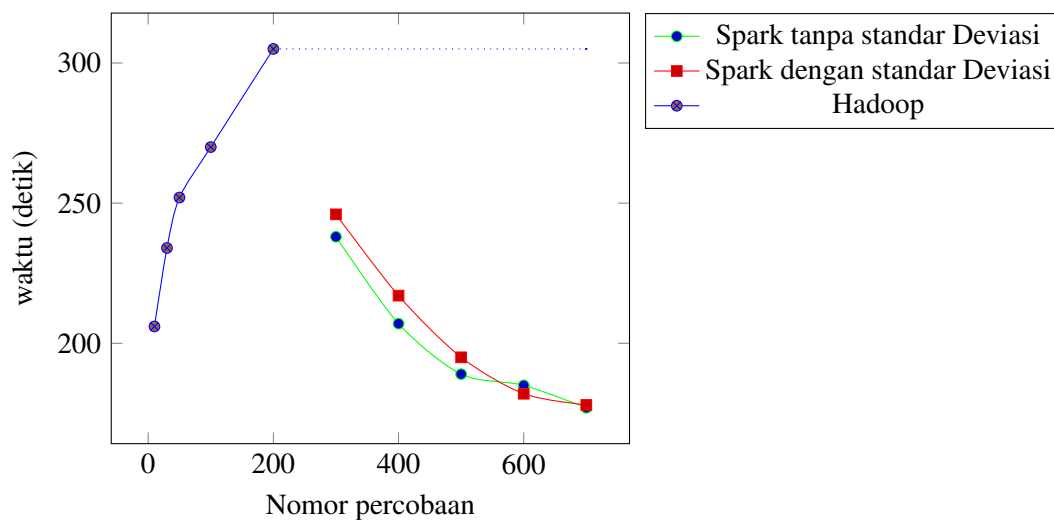
Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang optimal dengan jumlah objek maksimum adalah 30 untuk setiap *dendrogram*. Satu komputer akan digunakan sebagai komputer *master* dan 10 komputer lainnya sebagai *worker* dengan setiap *worker* menggunakan 5 core. Ukuran data yang digunakan adalah 5 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 30. Tabel (??) dan Tabel (5.37) berikut adalah hasil dari eksperimen:

Tabel 5.36: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 5 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (detik)	Hasil Reduksi Hadoop (GB)
5	10	206	2.2
5	30	234	2.2
5	50	252	2.2
5	100	270	2.2
5	200	305	2.2

Tabel 5.37: Percobaan Jumlah Partisi Spark dengan Ukuran Data 5 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark standar (detik)	Waktu Eksekusi Tanpa Deviasi	Waktu Eksekusi Spark (detik)	Hasil Spark standar (GB)	Hasil Tanpa Deviasi	Hasil Reduksi Spark (GB)	Hasil Reduksi
5	300	238		246	2.1		2.6	
5	400	207		217	2.1		2.6	
5	500	189		195	2.1		2.6	
5	600	185		182	2.1		2.6	
5	700	177		178	2.1		2.6	



Gambar 5.25: Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 5 GB

Berdasarkan hasil grafik (5.25), dapat dilihat waktu Hadoop terus meningkat seiring meningkatnya jumlah partisi. Jumlah partisi yang dicoba pada Hadoop hanya mencapai 200 karena waktu eksekusi yang sudah berbeda jauh dibanding Spark dan untuk partisi yang lebih besar pasti diatas garis biru titik-titik sesuai pola peningkatan waktu eksekusi Hadoop terhadap jumlah partisi. Garis biru titik-titik merupakan waktu eksekusi Hadoop dengan jumlah partisi 200. Spark memiliki waktu eksekusi yang jauh lebih cepat dibanding Hadoop.

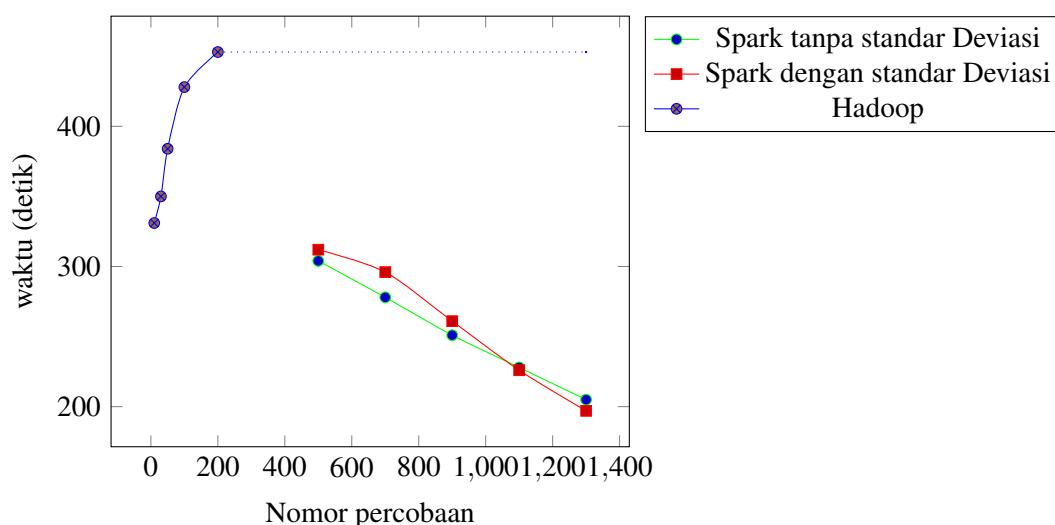
Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang optimal dengan jumlah objek maksimum adalah 30 untuk setiap *dendrogram*. Satu komputer akan digunakan sebagai komputer *master* dan 10 komputer lainnya sebagai *worker* dengan setiap *worker* menggunakan 5 core. Ukuran data yang digunakan adalah 10 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 30. Tabel (5.38) dan Tabel (5.39) berikut adalah hasil dari eksperimen:

Tabel 5.38: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 10 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (detik)	Hasil Reduksi Hadoop (GB)
10	10	331	3.9
10	30	350	3.9
10	50	384	3.9
10	100	428	3.9
10	200	453	3.9

Tabel 5.39: Percobaan Jumlah Partisi Spark dengan Ukuran Data 10 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark standar (detik)	Eksekusi Tanpa Deviasi	Waktu Eksekusi Spark (detik)	Hasil Reduksi Spark standar (GB)	Hasil Reduksi Spark (GB)
10	500	304		312	4.6	3.7
10	700	278		296	4.6	3.7
10	900	251		261	4.6	3.7
10	1100	228		226	4.6	3.7
10	1300	205		197	4.6	3.7



Gambar 5.26: Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 10 GB

Berdasarkan hasil grafik (5.26), dapat dilihat waktu Hadoop terus meningkat seiring meningkatnya jumlah partisi. Jumlah partisi yang dicoba pada Hadoop hanya mencapai 200 karena waktu eksekusi yang sudah berbeda jauh dibanding Spark dan untuk partisi yang lebih besar pasti diatas garis biru titik-titik sesuai pola peningkatan waktu eksekusi Hadoop terhadap jumlah partisi. Garis biru titik-titik merupakan waktu eksekusi Hadoop dengan jumlah partisi 200. Spark memiliki waktu eksekusi yang jauh lebih cepat dibanding Hadoop.

Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang optimal dengan jumlah objek maksimum adalah 30 untuk setiap *dendrogram*. Satu komputer akan digunakan sebagai komputer *master* dan 10 komputer lainnya sebagai *worker* dengan setiap *worker* munggunakan 5 core.

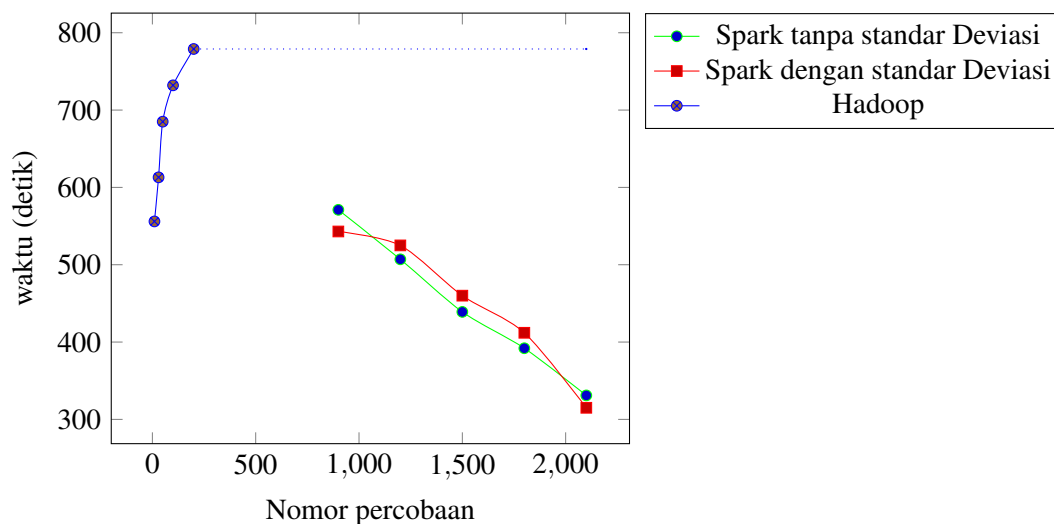
- 1 Ukuran data yang digunakan adalah 15 GB. Metode yang digunakan adalah metode *single linkage*, dengan
 2 nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 30. Tabel
 3 (5.40) dan Tabel (5.41) berikut adalah hasil dari eksperimen:

Tabel 5.40: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 15 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (detik)	Hasil Reduksi Hadoop (GB)
15	10	556	6.1
15	30	613	6.1
15	50	685	6.1
15	100	732	6.1
15	200	779	6.1

Tabel 5.41: Percobaan Jumlah Partisi Spark dengan Ukuran Data 15 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark standar (detik)	Eksekusi Tanpa Deviasi	Waktu Eksekusi Spark (detik)	Hasil Reduksi Spark standar (GB)	Hasil Reduksi Tanpa Deviasi	Hasil Reduksi Spark (GB)
15	900	471		443	5.8		7.3
15	1200	407		425	5.8		7.3
15	1500	339		360	5.8		7.3
15	1800	292		282	5.8		7.3
15	2100	231		245	5.8		7.3



Gambar 5.27: Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 15 GB

- 5 Berdasarkan hasil grafik (5.27), dapat dilihat waktu Hadoop terus meningkat seiring meningkatnya
 6 jumlah partisi. Jumlah partisi yang dicoba pada Hadoop hanya mencapai 200 karena waktu eksekusi yang
 7 sudah berbeda jauh dibanding Spark dan untuk partisi yang lebih besar pasti diatas garis biru titik-titik sesuai
 8 pola peningkatan waktu eksekusi Hadoop terhadap jumlah partisi. Garis biru titik-titik merupakan waktu
 9 eksekusi Hadoop dengan jumlah partisi 200. Spark memiliki waktu eksekusi yang jauh lebih cepat dibanding
 10 Hadoop.
 11

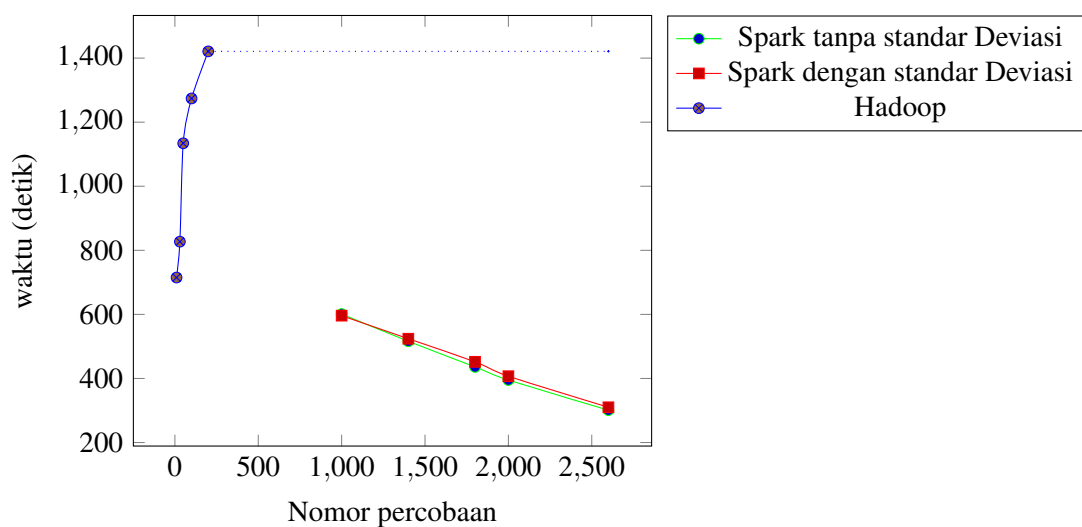
Pada percobaan ini akan dilihat waktu eksekusi Spark dan Hadoop berdasarkan jumlah partisi yang optimal dengan jumlah objek maksimum adalah 30 untuk setiap *dendrogram*. Satu komputer akan digunakan sebagai komputer *master* dan 10 komputer lainnya sebagai *worker* dengan setiap *worker* menggunakan 5 core. Ukuran data yang digunakan adalah 20 GB. Metode yang digunakan adalah metode *single linkage*, dengan nilai *cut-off distance* adalah 0,8 dan jumlah objek maksimum untuk setiap *dendrogram* adalah 30. Tabel (5.42) dan Tabel (5.43) berikut adalah hasil dari eksperimen:

Tabel 5.42: Percobaan Jumlah Partisi Hadoop dengan Ukuran Data 20 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Hadoop (detik)	Hasil Reduksi Hadoop (GB)
20	10	715	8.1
20	30	827	8.1
20	50	1134	8.1
20	100	1274	8.1
20	200	1421	8.1

Tabel 5.43: Percobaan Jumlah Partisi Spark dengan Ukuran Data 20 GB

Ukuran Data (GB)	Jumlah Partisi	Waktu Eksekusi Spark Tanpa standar Deviasi (detik)	Waktu Eksekusi Spark (detik)	Hasil Reduksi Spark standar (GB)	Hasil Reduksi Spark Tanpa standar Deviasi (GB)
20	1000	601	596	7.7	9.6
20	1400	516	524	7.7	9.6
20	1800	436	452	7.7	9.6
20	2200	395	407	7.7	9.6
20	2600	301	310	7.7	9.6



Gambar 5.28: Hasil Percobaan Jumlah Partisi Spark dan Hadoop dengan Ukuran Data 20 GB

Berdasarkan hasil grafik (5.28), dapat dilihat waktu Hadoop terus meningkat seiring meningkatnya jumlah partisi. Jumlah partisi yang dicoba pada Hadoop hanya mencapai 200 karena waktu eksekusi yang

- 1 sudah berbeda jauh dibanding Spark dan untuk partisi yang lebih besar pasti diatas garis biru titik-titik sesuai
- 2 pola peningkatan waktu eksekusi Hadoop terhadap jumlah partisi. Garis biru titik-titik merupakan waktu
- 3 eksekusi Hadoop dengan jumlah partisi 200. Spark memiliki waktu eksekusi yang jauh lebih cepat dibanding
- 4 Hadoop.

BAB 6

KESIMPULAN DAN SARAN

Bab ini berisi kesimpulan dari awal hingga akhir penelitian beserta saran untuk penelitian selanjutnya.

6.1 Kesimpulan

Kesimpulan yang dapat ditarik dari awal penelitian ini sampai selesai adalah sebagai berikut:

- Pada penelitian ini, telah dipelajari algoritma *Hierarchical Agglomerative Clustering*.
- Pada penelitian ini, telah diimplementasikan algoritma *Hierarchical Agglomerative Clustering* pada lingkungan Spark.
- Pada penelitian ini, telah dilakukan eksperimen perbandingan performa antara perangkat lunak Spark dan Hadoop. Dari hasil pengujian dapat disimpulkan bahwa perangkat lunak Spark memiliki performa yang lebih baik. Perangkat lunak Spark memiliki waktu eksekusi yang lebih cepat dan dapat memproses data yang berukuran lebih besar dibanding Hadoop. Waktu eksekusi Spark lebih cepat dibanding Hadoop karena Spark menyimpan data pada memori, Sebaliknya Hadoop banyak melakukan proses I/O kepada disk yang membuat Hadoop lambat.
- Pada penelitian ini, telah dibangun perangkat lunak untuk melihat hasil reduksi data.

6.2 Saran

Saran untuk penelitian selanjutnya adalah sebagai berikut:

- Pada penelitian ini, Spark dijalankan pada Hadoop *YARN*. Oleh karena itu, penulis berharap agar penelitian selanjutnya dapat menguji performa perangkat lunak pada Spark *cluster*.
- Pada penelitian ini, jumlah partisi akan berdampak pada waktu eksekusi perangkat lunak Spark. Oleh karena itu, penulis berharap agar penelitian selanjutnya dapat meneliti dampak banyaknya partisi terhadap waktu eksekusi.
- Pada penelitian ini, pengujian yang dilakukan masih terbatas dengan 10 *worker* dan ukuran data sampai 20GB. Untuk penelitian selanjutnya, penulis berharap agar pengujian yang dilakukan dapat menggunakan jumlah *worker* dan data yang lebih besar.

DAFTAR REFERENSI

- [1] Ishwarappa dan J, A. (2015) A brief introduction on big data 5vs characteristics and hadoop technology. *Procedia Computer Science*, **48**, 319 – 324.
- [2] Moertini, V. S., Suarjana, G. W., Venica, L., dan Karya, G. (2018) Big data reduction technique using parallel hierarchical agglomerative clustering. *IAENG International Journal of Computer Science*, **45**, 188 – 205.
- [3] Jain, A. K. dan Dubes, R. C. (1988) *Algorithms for Clustering Data*. Pearson College Div, New Jersey.
- [4] Holmes, A. (2012) *Hadoop in Practice*. Manning, New York.
- [5] White, T. (2015) *Hadoop The Definitive Guide*, 4th edition. O'Reilly Media, Sebastopol.
- [6] Lam, C. (2010) *Hadoop in Action*. Manning Publications, New York.
- [7] Karau, H., Konwinski, A., Wndell, P., dan Zaharia, M. (2015) *Learning Spark*, 1th edition. O'Reilly Media, Sebastopol.

LAMPIRAN A

KODE PROGRAM

Listing A.1: Main.scala

```
1 package main.scala
2 import org.apache.spark.{SparkConf, SparkContext}
3
4 object Main {
5   def main(args: Array[String]): Unit = {
6     val master = "yarn-cluster"
7     val input = args(0)
8     val output = args(1)
9     val numPar = args(2).toInt
10    val maxObj = args(3).toInt
11    val distType = args(4).toInt
12    val cutOffDist = args(5).toDouble
13    val conf = new SparkConf()
14    conf.setMaster(master)
15    conf.setAppName("Reduce_Data_Spark")
16    val sc = new SparkContext(conf)
17    val dataReducer = new DataReducer(sc, numPar, maxObj, distType, cutOffDist, input, output)
18    dataReducer.reduceData()
19  }
20 }
```

Listing A.2: DataReducer.scala

```
1 package main.scala
2
3 import org.apache.spark.{SparkContext}
4 import org.apache.spark.rdd.RDD
5 import scala.collection.mutable.{ListBuffer}
6
7 class DataReducer(sc: SparkContext, numPar: Int, maxObj: Int, distanceType: Int, cutOffDistance: Double, inputPath: String, outputPath: String) extends Serializable {
8
9   def reduceData(): Unit = {
10     val data = mapData()
11     val broadCastMaxObj = sc.broadcast(maxObj)
12     val broadCastDistanceType = sc.broadcast(distanceType)
13     val broadCastCutOffDist = sc.broadcast(cutOffDistance)
14     val results = data.mapPartitions(partitions => {
15       val patterns: ListBuffer[Patern] = new ListBuffer[Patern]()
16       var i: Int = 0
17       var isProcessed: Boolean = false
18       var objectList: ListBuffer[Node] = new ListBuffer[Node]()
19       partitions.foreach(record => {
20         isProcessed = false
21         i += 1
22         objectList += record
23         if (i == broadCastMaxObj.value) {
24           val dendrogram: Dendrogram = new Dendrogram(objectList, broadCastDistanceType.value)
25           dendrogram.generateDendrogram()
26           val cluster = new Cluster(dendrogram.getDendrogram(), broadCastCutOffDist.value)
27           patterns = patterns ++ cluster.computePatern()
28           isProcessed = true
29           i = 0
30           objectList.clear()
31         }
32       })
33       if (isProcessed == false) {
34         val dendrogram: Dendrogram = new Dendrogram(objectList, broadCastDistanceType.value)
35         dendrogram.generateDendrogram()
36         val cluster = new Cluster(dendrogram.getDendrogram(), broadCastCutOffDist.value)
37         patterns = patterns ++ cluster.computePatern()
38       }
39       patterns.toIterator
40     })
41     results.map(x => x.getObjCount() + "\n"
42       + x.getMinArr().mkString(", ") + "\n"
43       + x.getMaxArr().mkString(", ") + "\n"
44       + x.getAvgArr().mkString(", ") + "\n"
45       + x.getSDArr().mkString(", "))
46     ).saveAsTextFile(outputPath)
47     broadCastMaxObj.destroy()
48     broadCastDistanceType.destroy()
49     broadCastCutOffDist.destroy()
50     System.in.read()
51   }
```

```

51     sc.stop()
52 }
53
54 private def loadData():RDD[String] = {
55     sc.textFile(inputPath, numPar)
56 }
57
58 private def mapData():RDD[Node] = {
59     loadData().map(lines => {
60         val node = new Node()
61         node.setData(lines.split(",").map(_.toDouble))
62         node
63     })
64 }
65 }

```

Listing A.3: Dendrogram.scala

```

1 package main.scala
2
3 import scala.collection.mutable.{ArrayBuffer, ListBuffer}
4
5 class Dendrogram(nodeList:ListBuffer[Node], distType:Int) extends Serializable {
6     private var dendrogram = new ArrayBuffer[Node]()
7     private var nodeListCluster= new ArrayBuffer[ListBuffer[Node]]()
8     private var distanceMatrix = new ArrayBuffer[ArrayBuffer[Double]]()
9
10    def getDendrogram(): Node = {
11        dendrogram(0)
12    }
13
14    def generateDendrogram(): Unit ={
15        var i = 0
16        nodeList.foreach(node => {
17            dendrogram += node
18            nodeListCluster += new ListBuffer[Node]
19            nodeListCluster(i) += node
20            distanceMatrix += new ArrayBuffer[Double]()
21            i+=1
22        })
23        i = 1
24        var x = 0
25        for(i <- 1 until distanceMatrix.length){
26            for(x <- 0 until i){
27                distanceMatrix(i) += findMinimumDistance(nodeListCluster(i),nodeListCluster(x))
28            }
29        }
30
31        while(dendrogram.length !=1){
32            var x = 1
33            var y = 0
34            var result = Double.MaxValue
35            var coordinateX = 0
36            var coordinateY = 0
37            var temp = 0.0
38            for(x <- 1 until distanceMatrix.length){
39                for(y <- 0 until x){
40                    temp = distanceMatrix(x)(y)
41                    if(temp < result){
42                        result = temp
43                        coordinateX = x
44                        coordinateY = y
45                    }
46                }
47            }
48            formClusterBetweenNearestNeighbour(coordinateX,coordinateY)
49            recalculateMatrix(coordinateX,coordinateY)
50        }
51    }
52
53    private def formClusterBetweenNearestNeighbour(x:Int,y:Int): Unit = {
54        nodeListCluster(y) = nodeListCluster(y) ++ nodeListCluster(x)
55        nodeListCluster.remove(x)
56        val cluster = new Node()
57        cluster.setDistance(distanceMatrix(x)(y))
58        cluster.setLeftNode(dendrogram(y))
59        cluster.setRightNode(dendrogram(x))
60        dendrogram(y) = cluster
61        dendrogram.remove(x)
62    }
63
64
65    private def recalculateMatrix(x:Int,y:Int): Unit ={
66        distanceMatrix.remove(x)
67        for(i <- x+1 until distanceMatrix.length){
68            distanceMatrix(i).remove(x)
69        }
70        for(i <- y+1 until distanceMatrix.length){
71            distanceMatrix(i)(y) = findMinimumDistance(nodeListCluster(i), nodeListCluster(y))
72        }
73    }
74
75    private def findMinimumDistance(firstList:ListBuffer[Node],secondList:ListBuffer[Node]): Double ={
76        if(distType == 0) calculateSingleLinkage(firstList,secondList)
77        else if (distType == 1) calculateCompleteLinkage(firstList, secondList)
78        else calculateCentroidLinkage(firstList,secondList)
79    }
80

```

```

81 | private def calculateCentroidLinkage(firstList:ListBuffer[Node], secondList:ListBuffer[Node]): Double = {
82 |     val length = firstList(0).getData().length
83 |     val firstArr = new Array[Double](length)
84 |     val secondArr = new Array[Double](length)
85 |     var i = 0
86 |     var max = firstList.length
87 |     if(secondList.length > max) max = secondList.length
88 |     while(i < max){
89 |         if(i < firstList.length ){
90 |             var index = 0;
91 |             firstList(i).getData().foreach( data => {
92 |                 firstArr(index) += data
93 |                 index+=1
94 |             })
95 |         }
96 |         if(i < secondList.length){
97 |             var index = 0;
98 |             secondList(i).getData().foreach( data => {
99 |                 secondArr(index) += data
100 |                 index+=1
101 |             })
102 |         }
103 |         i+=1
104 |     }
105 |     i=0
106 |     while(i<firstArr.length){
107 |         firstArr(i) /= firstList.length
108 |         secondArr(i) /= secondList.length
109 |         i+=1
110 |     }
111 |     calculateDistance(firstArr,secondArr)
112 | }
113 |
114 | private def calculateSingleLinkage(firstList:ListBuffer[Node], secondList:ListBuffer[Node]): Double = {
115 |     var min:Double = Double.MaxValue
116 |     var result:Double = 0
117 |     firstList.foreach( nodeA => {
118 |         secondList.foreach( nodeB => {
119 |             result = calculateDistance(nodeA.getData(), nodeB.getData())
120 |             if(result < min) min = result
121 |         })
122 |     })
123 |     min
124 | }
125 |
126 | private def calculateCompleteLinkage(firstList:ListBuffer[Node], secondList:ListBuffer[Node]): Double = {
127 |     var max:Double = Double.MinValue
128 |     var result:Double = 0
129 |     firstList.foreach( nodeA => {
130 |         secondList.foreach( nodeB => {
131 |             result = calculateDistance(nodeA.getData(), nodeB.getData())
132 |             if(result > max) max = result
133 |         })
134 |     })
135 |     max
136 | }
137 |
138 | private def calculateDistance(firstArr:Array[Double], secondArr:Array[Double]): Double = {
139 |     val n = firstArr.length-1
140 |     var total:Double = 0
141 |     for(i <- 0 to n){
142 |         total +=Math.pow(firstArr(i)-secondArr(i),2)
143 |     }
144 |     Math.sqrt(total)
145 | }
146 | }

```

Listing A.4: Cluster.scala

```

1 | package main.scala
2 |
3 | import scala.collection.mutable.{ArrayBuffer, ListBuffer}
4 |
5 | class Cluster(dendrogram:Node, cutOffDistance:Double) extends Serializable {
6 |     private val clusters:ListBuffer[Node] = new ListBuffer[Node]()
7 |
8 |     private def formClusterFromDendrogram(): Unit = {
9 |         val bfs:ListBuffer[Node] = new ListBuffer[Node]
10 |         bfs+=dendrogram
11 |         val distance = cutOffDistance * dendrogram.getDistance()
12 |         while(bfs.length!=0){
13 |             var node = bfs.remove(0)
14 |             if(node.getDistance() <= distance){
15 |                 clusters+=node
16 |             } else {
17 |                 var left = node.getLeftNode()
18 |                 var right = node.getRightNode()
19 |                 if(left!=null){
20 |                     bfs+=left
21 |                 }
22 |                 if(right!=null){
23 |                     bfs+=right
24 |                 }
25 |             }
26 |         }
27 |     }
28 |
29 |     def computePatern(): ListBuffer[Patern] = {

```

```

30     formClusterFromDendrogram()
31     val patterns: ListBuffer[Patern] = new ListBuffer[Patern]()
32     clusters.foreach( cluster => {
33         patterns += processCluster(cluster)
34     })
35     patterns
36 }
37
38 private def processCluster(cluster: Node): Patern = {
39     val bfs: ListBuffer[Node] = new ListBuffer[Node]()
40     val min: ArrayBuffer[Double] = new ArrayBuffer[Double]()
41     val max: ArrayBuffer[Double] = new ArrayBuffer[Double]()
42     val avg: ArrayBuffer[Double] = new ArrayBuffer[Double]()
43     val SD: ArrayBuffer[Double] = new ArrayBuffer[Double]()
44     bfs += cluster
45     var count = 0
46     var i = 0
47     while(bfs.length != 0) {
48         val node = bfs.remove(0)
49         val data = node.getData()
50         if(data != null) {
51             if(min.length == 0) {
52                 data.foreach(value => {
53                     min += value
54                     max += value
55                     avg += value
56                 })
57             } else {
58                 i = 0
59                 data.foreach(value => {
60                     if(value < min(i)) min(i) = value
61                     if(value > max(i)) max(i) = value
62                     avg(i) += value
63                     i += 1
64                 })
65             }
66             count += 1
67         } else {
68             val leftNode = node.getLeftNode()
69             val rightNode = node.getRightNode()
70             if(leftNode != null) {
71                 bfs += leftNode
72             }
73             if(rightNode != null) {
74                 bfs += rightNode
75             }
76         }
77     }
78     i = 0
79     avg.foreach( value => {
80         avg(i) /= count
81         i += 1
82     })
83     bfs += cluster
84     while(bfs.length != 0) {
85         val node = bfs.remove(0)
86         val data = node.getData()
87         if(data != null) {
88             if(SD.length == 0) {
89                 i = 0
90                 data.foreach(value => {
91                     //println("TEST SD")
92                     //println(value + " " + avg(i))
93                     SD += Math.pow((value - avg(i)), 2)
94                     //println(SD(0))
95                     i += 1
96                 })
97             } else {
98                 i = 0
99                 data.foreach(value => {
100                     SD(i) += Math.pow((value - avg(i)), 2)
101                     i += 1
102                 })
103             }
104         } else {
105             val leftNode = node.getLeftNode()
106             val rightNode = node.getRightNode()
107             if(leftNode != null) {
108                 bfs += leftNode
109             }
110             if(rightNode != null) {
111                 bfs += rightNode
112             }
113         }
114     }
115     i = 0
116     SD.foreach( value => {
117         if(count == 1) {
118             SD(i) = 0
119         } else {
120             SD(i) = Math.sqrt((SD(i) / (count - 1)))
121             i += 1
122         }
123     })
124     new Patern(max.toArray, min.toArray, avg.toArray, SD.toArray, count)
125 }
126 }

```

Listing A.5: Node.scala

```

1 package main.scala
2
3 class Node() extends Serializable {
4     private var data:Array[Double] = null
5     private var distance:Double = -1
6     private var rightNode:Node = null
7     private var leftNode:Node = null
8
9     def setData(data: Array[Double]): Unit = {
10         this.data = data
11     }
12
13     def setDistance(distance: Double): Unit = {
14         this.distance = distance
15     }
16
17     def setRightNode(node:Node): Unit = {
18         this.rightNode = node
19     }
20
21     def setLeftNode(node:Node): Unit = {
22         this.leftNode = node
23     }
24
25     def getData(): Array[Double] = {
26         this.data
27     }
28
29     def getDistance():Double = {
30         this.distance
31     }
32
33     def getRightNode(): Node = {
34         this.rightNode
35     }
36
37     def getLeftNode(): Node = {
38         this.leftNode
39     }
40 }

```

Listing A.6: Node.scala

```

1 package main.scala
2
3 class Patern(max:Array[Double], min:Array[Double], avg:Array[Double], SD:Array[Double], objCount:Int) extends Serializable {
4
5     def getMaxArr(): Array[Double] = {
6         max
7     }
8
9     def getMinArr(): Array[Double] = {
10         min
11     }
12
13     def getAvgArr(): Array[Double] = {
14         avg
15     }
16
17     def getSDArr(): Array[Double] = {
18         SD
19     }
20
21     def getObjCount(): Int = {
22         objCount
23     }
24 }

```


LAMPIRAN B

KODE PROGRAM UNTUK ANTARMUKA

Listing B.1: index.php

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <?php include 'head.php'; ?>
5   <title>Spark Reduce Data App UI</title>
6   <style type="text/css">
7     .input-field label {
8       color: #212121;
9     }
10    /* label focus color */
11    .input-field input[type=text]:focus + label {
12      color: #212121;
13    }
14    /* label underline focus color */
15    .input-field input[type=text]:focus {
16      border-bottom: 1px solid #212121;
17      box-shadow: 0 1px 0 0 #212121;
18    }
19    /* icon prefix focus color */
20    .input-field .prefix.active {
21      color: #212121;
22    }
23  </style>
24 </head>
25
26 <body>
27   <?php include 'nav.php' ?>
28   <div class="row">
29     <div class="col_m4_offset-m4">
30       <div class="row">
31         <div class="col_m12" style="margin-top:_40px;">
32           <div class="card_teal_lighten-5">
33             <div class="card-content_black-text">
34               <span class="card-title">Submit Spark Application</span>
35             <div class="row">
36               <?php include 'form.php'; ?>
37             </div>
38           </div>
39         </div>
40       </div>
41     </div>
42   </div>
43 </body>
44 </html>
45 <script type="text/javascript">
46   $(document).ready(function(){
47
48     $('select').formSelect();
49
50     $( "#spark-form" ).submit(function( event ) {
51       window.open( 'master:8080/cluster', '_blank' );
52     });
53   });
54 </script>
```

Listing B.2: head.php

```
1 <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/css/materialize.min.css">
2 <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.0/jquery.min.js"></script>
3 <script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/js/materialize.min.js"></script>
```

Listing B.3: data.php

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <?php include 'head.php' ?>
5   <title>Spark Reduce Data App UI</title>
6   <style type="text/css">
7   </style>
8 </head>
9
```

```

10 <body>
11 <?php include 'nav.php' ?>
12 <div class="row">
13 <div class="col">
14 <div class="row">
15 <div class="col_m12" style="margin-top:10px;">
16
17 <?php
18 $temp = "00000";
19 $sub = substr($temp, strlen($_GET['part']));
20 //echo $sub;
21 //echo $_GET['part'];
22 $output = shell_exec('cd_/home/miebakso/hadoop-2.7.3_&&_/bin/hadoop_fs_cat_' . $_GET['path'] . '/part-' . $sub .
    $_GET['part']);
23 $arr = explode("\n", $output);
24 $i = 1;
25 $len = count(explode(',', $arr[1]));
26 foreach ($arr as $value) {
27     if($i==1){
28         echo '
29         <div class="card_teal_lighten-4">
30         <div class="card-content_black-text">
31         <table>
32         <thead>
33         <tr>
34         <th>Total_Obj_ ' . $value . ' </th>';
35         $x=1;
36         while($x<=$len){
37             echo ' <th>attribute- ' . $x . ' </th>';
38             $x++;
39         }
40
41         echo '
42         </tr>
43         </thead>
44
45         <tbody>
46         '
47         ;
48
49     } else if($i==2){
50         $arr2 = explode(",", $value);
51         echo ' <tr><td>Minimum</td>';
52         foreach ($arr2 as $val) {
53             echo ' <td>' . $val . ' </td>';
54         }
55         echo ' </tr>';
56     } else if($i==3){
57         $arr2 = explode(",", $value);
58         echo ' <tr><td>Maximum</td>';
59         foreach ($arr2 as $val) {
60             echo ' <td>' . $val . ' </td>';
61         }
62         echo ' </tr>';
63     } else if($i==4){
64         $arr2 = explode(",", $value);
65         echo ' <tr><td>Average</td>';
66         foreach ($arr2 as $val) {
67             echo ' <td>' . $val . ' </td>';
68         }
69         echo ' </tr>';
70     } else {
71         $arr2 = explode(",", $value);
72         echo ' <tr><td>Stardard_Deviation</td>';
73         foreach ($arr2 as $val) {
74             echo ' <td>' . $val . ' </td>';
75         }
76         echo ' </tr>
77
78         </tbody>
79         </table>
80         </div>
81         </div>
82         '
83         ;
84         $i=0;
85     }
86     $i+=1;
87 }
88
89 </div>
90 </div>
91 </div>
92 </div>
93 </body>
94 </html>
95 <script type="text/javascript">
96 $(document).ready(function(){
97
98     $( "#spark-data" ).submit(function( event ) {
99         window.open( 'localhost:50070/explorer.html#' + $( '#data_path' ).val(), '_blank' );
100     });
101 });
102 </script>

```

Listing B.4: nav.php


```

2 <div class="nav-wrapper_blue">
3 <ul id="nav-mobile" class="left_hide-on-med-and-down">
4 <li><a href="index.php" style="font-size:_40px;">Submit</a></li>
5 <li><a href="view.php" style="font-size:_40px;">Patern</a></li>
6 </ul>
7 </div>
8 </nav>

```

Listing B.5: form.php

```

1 <form id="spark-form" class="col_m12" method="post" action="result.php" style="font-size:_20px;">
2 <div class="row">
3 <div class="input-field_col_m12_s12_black-text">
4 <input id="jar_path" type="text" name="jar_path" class="validate">
5 <label for="jar_path" >Spark JAR Path</label>
6 </div>
7 </div>
8 <div class="row">
9 <div class="input-field_col_m12_s12_black-text">
10 <input id="input_path" type="text" name="input_path" class="black-text">
11 <label for="input_path" >Input Path</label>
12 </div>
13 </div>
14 <div class="row">
15 <div class="input-field_col_m12_s12_black-text">
16 <input id="output_path" type="text" name="output_path" class="black-text">
17 <label for="output_path" >Output Path</label>
18 </div>
19 </div>
20
21 <div class="row">
22 <div class="input-field_col_m4_s12_black-text">
23 <input id="number_of_executor" type="number" name="executor_number" class="black-text" value="1" min="1" step="1" max=
    "100">
24 <label for="number_of_executor" >Number of Executor</label>
25 </div>
26 <div class="input-field_col_m4_s12_black-text">
27 <input id="executor_memory" type="number" name="executor_memory" class="black-text" value="1000" min="1000" step="100"
    >
28 <label for="executor_memory" >Executor Memory in mb</label>
29 </div>
30 <div class="input-field_col_m4_s12_black-text">
31 <input id="number_of_partition" type="number" name="number" class="black-text" value="1" min="1" step="1" max="200">
32 <label for="number_of_partition" >Number of Partition</label>
33 </div>
34 </div>
35 <div class="row">
36 <div class="input-field_col_m4_s12_black-text">
37 <input id="max_obj" type="number" name="max_obj" class="black-text" value="1" min="1" step="1" max="100">
38 <label for="max_obj" >Max Object</label>
39 </div>
40 <div class="input-field_col_m4_s12_black-text">
41 <select name="type">
42 <option value="0" >Single Linkage</option>
43 <option value="1" >Complete Linkage</option>
44 <option value="2" >Centroid Linkage</option>
45 </select>
46 </div>
47 <div class="input-field_col_m4_s12_black-text">
48 <input id="cut_off" type="number" name="cut_off" class="black-text" value="0.1" min="0.1" step="0.1" max="1">
49 <label for="cut_off" >Cut Off Distance</label>
50 </div>
51 </div>
52
53 <button class="btn_waves-effect_waves-light" type="submit" name="action">Submit
54 <i class="material-icons_right"></i>
55 </button>
56 </form>

```

Listing B.6: list.php

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <?php include 'head.php' ?>
5 <title>Spark Reduce Data App UI</title>
6 <style type="text/css">
7 </style>
8 </head>
9
10 <body>
11 <?php include 'nav.php' ?>
12 <div class="row">
13 <div class="col_m3_offset-m1">
14 <div class="row">
15 <div class="col_m12" style="margin-top:_20px;">
16 <div class="card_teal_lighten-5">
17 <div class="card-content_black-text">
18 <?php
19 $output = shell_exec('cd_/home/miebakso/hadoop-2.7.3&&_/bin/hadoop_fs_ls_'. $_POST['data_path']);
20 $sarr = explode("\n", $output);
21 $len = count($sarr)-2;
22 $i=0;
23 while($i<$len){
24 <echo "<a href='data.php?part=". $i."&path=". $_POST['data_path']."'>part-". $i."</a><br>";
25 $i=$i+1;

```

```

26         }
27     ?>
28     </div>
29 </div>
30 </div>
31 </div>
32 </div>
33 </div>
34 </body>
35 </html>
36 <script type="text/javascript">
37     $(document).ready(function(){
38
39         $("#spark-data").submit(function( event ) {
40             window.open( 'localhost:50070/explorer.html#' + $('#data_path').val(), '_blank' );
41         });
42     });
43 </script>

```

Listing B.7: result.php

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4     <?php include 'head.php' ?>
5     <title>Spark Reduce Data App UI</title>
6     <style type="text/css">
7     </style>
8 </head>
9
10 <body>
11     <?php include 'nav.php' ?>
12     <div class="row">
13         <div class="col_m4_offset-m4">
14             <div class="row">
15                 <div class="col_m12" style="margin-top: 30px;">
16                     <div class="card_teal_lighten-5">
17                         <div class="card-content_black-text">
18                             <span class="card-title">Submit Successful</span>
19                             <div class="row">
20                                 <?php
21                                     $jar = $_POST['jar_path'];
22                                     $input = $_POST['input_path'];
23                                     $output = $_POST['output_path'];
24                                     $executor_number = $_POST['executor_number'];
25                                     $executor_memory = $_POST['executor_memory'];
26                                     $partition = $_POST['number'];
27                                     $max_obj = $_POST['max_obj'];
28                                     $type = $_POST['type'];
29                                     $cutoff = $_POST['cut_off'];
30
31                                     $output = shell_exec('cd $_SPARK_HOME_&&_./bin/spark-submit_-class_main.scala.Main_-master_yarn_/home/
32                                         miebakso/IdeaProjects/BigData/target/scala-2.11/bigdata_2.11-0.1.jar');
33                                     ?>
34                                 </div>
35                             </div>
36                         </div>
37                     </div>
38                 </div>
39             </div>
40 </body>
41 </html>

```

Listing B.8: view.php

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4     <?php include 'head.php' ?>
5     <title>Spark Reduce Data App UI</title>
6     <style type="text/css">
7     </style>
8 </head>
9
10 <body>
11     <?php include 'nav.php' ?>
12     <div class="row">
13         <div class="col_m6_offset-m3">
14             <div class="row">
15                 <div class="col_m12" style="margin-top: 200px;">
16                     <div class="card_teal_lighten-5">
17                         <div class="card-content_black-text">
18                             <span class="card-title">Explore HDFS Directory</span>
19                             <div class="row">
20                                 <form id="spark-data" class="col_m12" action="list.php" method="post">
21                                     <div class="row">
22                                         <div class="input-field_col_m12_black-text">
23                                             <input id="data_path" type="text" name="data_path" class="validate">
24                                             <label for="data_path">HDFS data path</label>
25                                         </div>
26                                     </div>
27
28                                     <button class="btn_waves-effect_waves-light" type="submit" name="action">Submit
29                                     <i class="material-icons_right"></i>
30                                 </button>

```

```
31 |         </form>
32 |     </div>
33 | </div>
34 | </div>
35 | </div>
36 | </div>
37 | </body>
38 | </html>
39 | <script type="text/javascript">
40 |     $(document).ready(function(){
41 |
42 |         $( "#spark-data" ).submit(function( event ) {
43 |             window.open( 'localhost:50070/explorer.html#'+$( '#data_path' ).val(), '_blank' );
44 |         });
45 |     });
46 | </script>
```