



**EVALUASI PERFORMA HADOOP DAN SPARK PADA  
DIGITALOCEAN MENGGUNAKAN HIBENCH DALAM  
KONFIGURASI *PSEUDO DISTRIBUTED***

**NASKAH SKRIPSI**

**Dimas Wahyu Saputro  
NIM 120450081**

**PROGRAM STUDI SAINS DATA  
FAKULTAS SAINS  
INSTITUT TEKNOLOGI SUMATERA  
LAMPUNG SELATAN**

**2024**



**EVALUASI PERFORMA HADOOP DAN SPARK PADA  
DIGITALOCEAN MENGGUNAKAN HIBENCH DALAM  
KONFIGURASI *PSEUDO DISTRIBUTED***

**NASKAH SKRIPSI**  
**Diajukan sebagai syarat maju seminar hasil**

**Dimas Wahyu Saputro**  
**NIM 120450081**

**PROGRAM STUDI SAINS DATA**  
**FAKULTAS SAINS**  
**INSTITUT TEKNOLOGI SUMATERA**  
**LAMPUNG SELATAN**

**2024**

## **HALAMAN PENGESAHAN**

Naskah Tugas Akhir untuk Seminar Hasil dengan judul "**Evaluasi Performa Hadoop dan Spark pada DigitalOcean menggunakan HiBench dalam Konfigurasi Pseudo Distributed**" adalah benar dibuat oleh saya sendiri dan belum pernah dibuat dan diserahkan sebelumnya, baik sebagian ataupun seluruhnya, baik oleh saya ataupun orang lain, baik di Institut Teknologi Sumatera maupun di institusi pendidikan lainnya.

Lampung Selatan, 16 Mei 2024

Penulis,



**Dimas Wahyu Saputro**  
**NIM 120450081**

Diperiksa dan disetujui oleh,

Pembimbing I

Pembimbing II

**Tirta Setiawan, S.Pd., M.Si.**  
**NIP. 199008222022031003**

**Riksa Meidy Karim, S.Kom., M.Si., M.Sc.**

Disahkan oleh,  
Koordinator Program Studi Sains Data  
Fakultas Sains  
Institut Teknologi Sumatera

**Tirta Setiawan, S.Pd., M.Si.**  
**NIP. 199102302020012003**

## **HALAMAN PERNYATAAN ORISINALITAS**

**Skripsi ini adalah karya saya sendiri dan semua sumber baik yang dikutip  
maupun yang dirujuk telah saya nyatakan benar.**

**Nama : Dimas Wahyu Saputro**

**NIM : 120450081**

**Tanda tangan :**

**Tanggal : 16 Mei 2024**

## **HALAMAN PERNYATAAN PERSETUJUAN PUBLIKASI UNTUK KEPENTINGAN AKADEMIS**

Sebagai civitas akademik Institut Teknologi Sumatera, saya yang bertanda tangan di bawah ini:

Nama : Dimas Wahyu Saputro  
NIM : 120450081  
Program Studi : Sains Data  
Fakultas : Sains  
Jenis karya : Skripsi

demi pengembangan ilmu pengetahuan, menyetujui untuk memberikan Hak Bebas Royalti Noneksklusif (*Non-Exclusive Royalty Free Right*) kepada Institut Teknologi Sumatera atas karya ilmiah saya yang berjudul:

### **Evaluasi Performa Hadoop dan Spark pada DigitalOcean menggunakan Hi-Bench dalam Konfigurasi *Pseudo Distributed***

beserta perangkat yang ada (jika diperlukan). Dengan Hak Bebas Royalti Noneksklusif ini Institut Teknologi Sumatera berhak menyimpan, mengalihmedia/formatkan, mengelola dalam bentuk pangkalan data (*database*), merawat, dan memublikasikan tugas akhir saya selama tetap mencantumkan nama saya sebagai penulis/pencipta dan sebagai pemilik Hak Cipta.

Demikian pernyataan ini saya buat dengan sebenarnya.

Dibuat di : Lampung Selatan  
Pada tanggal : 16 Mei 2024

Yang menyatakan (Dimas Wahyu Saputro)

## ABSTRAK

### Evaluasi Performa Hadoop dan Spark pada DigitalOcean menggunakan HiBench dalam Konfigurasi *Pseudo Distributed*

Dimas Wahyu Saputro (120450081)

Pembimbing I: Tirta Setiawan, S.Pd., M.Si.

Pembimbing II: Riksa Meidy Karim, S.Kom., M.Si., M.Sc.

Perkembangan teknologi informasi mendorong peningkatan volume data yang dihasilkan dan disimpan setiap harinya. Hal ini menuntut *platform* komputasi terdistribusi yang efisien dan *scalable* untuk memproses data dalam skala besar. Hadoop dan Spark merupakan dua *platform* populer yang menawarkan solusi untuk *Big Data*. Penelitian ini bertujuan untuk membandingkan kinerja Hadoop dan Spark dalam mengolah data besar pada *platform cloud* DigitalOcean dengan fokus pada beban kerja *word count* dan *sort*, yang merupakan dasar bagi banyak aplikasi *data science*. *Word count* digunakan dalam pembuatan *Bag-of-Words* (BoW) untuk pemrosesan teks, sedangkan *sort* penting dalam proses pembobotan TF-IDF. Kedua *platform* diuji menggunakan *benchmark* HiBench dengan variasi ukuran data mulai dari 100 KB hingga 15 GB. Hasil penelitian menunjukkan Spark mampu menyelesaikan tugas *sort* dan *word count* dengan waktu eksekusi yang jauh lebih cepat, khususnya pada data berukuran besar. Pada beban kerja *sort*, Spark unggul mulai dari ukuran data 5 GB. Pada beban kerja *word count*, Spark unggul mulai dari ukuran data 500 MB. Secara keseluruhan, Spark menunjukkan kinerja yang lebih baik dalam menangani data berukuran besar, sementara Hadoop lebih efisien untuk data berukuran kecil hingga menengah. Spark juga lebih efisien dalam memanfaatkan CPU dan memori, serta meminimalkan operasi *disk I/O*. Hal ini menjadikan Spark *platform* yang lebih *scalable* dan efisien untuk pemrosesan data besar dibandingkan Hadoop, terutama untuk tugas *word count* dan *sort* yang menjadi fondasi bagi banyak aplikasi *data science*. Temuan ini diharapkan dapat memberikan panduan bagi para praktisi dalam memilih *platform* yang tepat untuk kebutuhan pemrosesan data.

**Kata kunci:** *Big data*, Hadoop, HiBench, Komputasi awan, Pemrosesan terdistribusi paralel, *Pseudo distributed*, Spark

## ***ABSTRACT***

### ***Performance Evaluation of Hadoop and Spark on DigitalOcean using HiBench in a Pseudo-Distributed Configuration***

Dimas Wahyu Saputro (120450081)

*Advisor I : Tirta Setiawan, S.Pd., M.Si.*

*Advisor II: Riksa Meidy Karim, S.Kom., M.Si., M.Sc.*

*The rapid advancement of information technology has led to an exponential increase in the volume of data generated and stored daily. This surge demands efficient and scalable distributed computing platforms to process large-scale data effectively. Hadoop and Spark are two widely adopted platforms that offer solutions for Big Data processing. This study aims to compare the performance of Hadoop and Spark in processing large datasets on the DigitalOcean cloud platform, focusing on the word count and sort workloads, which are foundational for numerous data science applications. Word count is utilized in constructing Bag-of-Words (BoW) for text processing, while sort plays a crucial role in TF-IDF weighting. Both platforms were tested using the HiBench benchmark with varying data sizes ranging from 100 KB to 15 GB. The findings reveal that Spark significantly outperforms Hadoop in terms of execution time, particularly for large datasets. While Hadoop demonstrates efficiency with smaller to medium-sized datasets, Spark excels in handling larger data volumes. In the sort workload, Spark consistently outperforms Hadoop starting from 5 GB of data. Similarly, in the word count workload, Spark's superiority is evident with data sizes exceeding 500 MB. Furthermore, Spark proved to be more efficient in utilizing CPU and memory resources, while minimizing disk I/O operations. These findings establish Spark as a more scalable and efficient platform for large-scale data processing compared to Hadoop, particularly for word count and sort tasks, which form the bedrock of many data science applications. This study provides valuable insights to guide practitioners in selecting the most suitable platform for their data processing requirements.*

***Keywords :*** *Big Data, Cloud Computing, Hadoop, HiBench, Parallel and Distributed Processing, Pseudo distributed, Spark.*

## MOTTO

*Urip iku Urup.*

## **HALAMAN PERSEMBAHAN**

*Untuk diriku, Ibu, dan Bapak.*

## **KATA PENGANTAR**

Puji syukur penulis ucapkan ke hadirat Allah SWT atas berkah dan rahmat-Nya sehingga skripsi ini dapat terselesaikan dengan baik. Skripsi ini merupakan karya yang wajib dibuat oleh mahasiswa untuk menyelesaikan pendidikan sarjana di Institut Teknologi Sumatera. Penyusunan skripsi ini banyak mendapat bantuan dan dukungan dari berbagai pihak sehingga dalam kesempatan ini, dengan penuh ke-rendahan hati, penulis mengucapkan terima kasih kepada:

1. Keluarga, Ibu Siti Ervingati dan bapak Kustriyanto, yang selalu memberikan doa, semangat, dukungan, dan motivasi sehingga penulis dapat mencapai tahap ini. Tak lupa pula untuk Dika, Habib, dan Syifa.
2. Bapak Tirta Setiawan, S.Pd., M.Si., selaku Koordinator Program Studi Sains Data Fakultas Sains Institut Teknologi Sumatera dan Dosen Pembimbing Utama.
3. Bapak Riksa Meidy Karim, S.Kom., M.Si., M.Sc., dan Ibu Amalya Citra S.Kom., M.Si., M.Sc., selaku dosen pembimbing pendamping yang telah memberikan arahan, ilmu, motivasi, serta saran kepada penulis.
4. Seluruh dosen dan tenaga kependidikan Sains Data Institut Teknologi Sumatera yang telah memberikan banyak bantuan dan ilmu selama penulis berku-liah.
5. Abil, Imam, Sakul, dan sahabat-sahabat yang tidak dapat disebutkan satu per-satu. Terima kasih atas semangat, bantuan dan motivasinya. Semoga kalian selalu dikuatkan.
6. Alfianri Manihuruk, teman-teman seperbimbingan, serta angkatan 2020 Sains Data Institut Teknologi Sumatera.

Penulis menyadari bahwa masih terdapat banyak kekurangan pada penulisan skripsi ini. Oleh karena itu, penulis mengharapkan kritik dan saran yang membangun dari pembaca demi perbaikan laporan ini. Semoga karya ini dapat bermanfaat bagi para pembaca pada umumnya dan juga bagi penulis pada khususnya.

Lampung Selatan, 16 Mei 2024

**Dimas Wahyu Saputro**

## DAFTAR ISI

<b>HALAMAN JUDUL . . . . .</b>	<b>i</b>
<b>HALAMAN PENGESAHAN . . . . .</b>	<b>ii</b>
<b>HALAMAN PERNYATAAN ORISINALITAS . . . . .</b>	<b>iii</b>
<b>HALAMAN PERNYATAAN PERSETUJUAN PUBLIKASI . . . . .</b>	<b>iv</b>
<b>ABSTRAK . . . . .</b>	<b>v</b>
<b>ABSTRACT . . . . .</b>	<b>vi</b>
<b>MOTTO . . . . .</b>	<b>vii</b>
<b>HALAMAN PERSEMBAHAN . . . . .</b>	<b>viii</b>
<b>KATA PENGANTAR . . . . .</b>	<b>ix</b>
<b>DAFTAR ISI . . . . .</b>	<b>x</b>
<b>DAFTAR GAMBAR . . . . .</b>	<b>xiii</b>
<b>DAFTAR TABEL . . . . .</b>	<b>xv</b>
<b>I PENDAHULUAN . . . . .</b>	<b>1</b>
1.1 Latar Belakang . . . . .	1
1.2 Rumusan Masalah . . . . .	3
1.3 Tujuan . . . . .	3
1.4 Batasan Masalah . . . . .	3
<b>II LANDASAN TEORI . . . . .</b>	<b>4</b>
2.1 Tinjauan Pustaka . . . . .	4
2.2 Konsep <i>Big Data</i> . . . . .	5
2.3 Ekstraksi Fitur Teks ( <i>Text Feature Extraction</i> ) . . . . .	5
2.3.1 <i>Bag of Words</i> (BoW) . . . . .	6
2.3.2 <i>Term Frequency-Inverse Document Frequency</i> (TF-IDF) . . . . .	6
2.3.3 Penggunaan <i>Word Count</i> dan <i>Sort</i> pada BoW dan TF-IDF . . . . .	7
2.4 Komputasi Awan ( <i>Cloud Computing</i> ) . . . . .	7
2.5 <i>Shell Script</i> . . . . .	8
2.6 MapReduce . . . . .	9
2.6.1 Apache Hadoop . . . . .	10
2.6.2 Mode Kerja Hadoop . . . . .	10
2.6.3 Hadoop Distributed File System (HDFS) . . . . .	11
2.6.4 Hadoop YARN . . . . .	12

2.7	Apache Spark . . . . .	13
2.7.1	Arsitektur Spark . . . . .	13
2.7.2	Integrasi Hadoop dan Spark . . . . .	14
2.7.3	Keterbatasan <i>Data Sharing</i> pada MapReduce . . . . .	15
2.7.4	Solusi <i>Data Sharing</i> dengan Spark RDD . . . . .	15
2.8	HiBench . . . . .	16
2.8.1	Beban Kerja <i>Micro Benchmark</i> dan Sumber Data . . . . .	17
2.8.2	<i>Data Generation</i> pada <i>Word Count</i> dan <i>Sort</i> . . . . .	18
2.8.3	Beban Kerja <i>Word Count</i> . . . . .	19
2.8.4	Beban Kerja <i>Sort</i> . . . . .	19
2.9	Data Keluaran HiBench dan Dool . . . . .	21
<b>III</b>	<b>METODOLOGI PENELITIAN . . . . .</b>	<b>23</b>
3.1	Alur Penelitian . . . . .	23
3.2	Identifikasi Masalah dan Studi Literatur . . . . .	24
3.3	Membangun <i>Virtual Machine</i> di DigitalOcean . . . . .	24
3.4	Pemasangan dan Konfigurasi Perangkat Lunak . . . . .	25
3.4.1	Instalasi Perangkat Lunak Prasyarat . . . . .	27
3.4.2	Instalasi dan Konfigurasi Hadoop . . . . .	27
3.4.3	Instalasi dan Konfigurasi Spark . . . . .	28
3.4.4	Instalasi dan Konfigurasi HiBench . . . . .	29
3.5	Eksperimen . . . . .	30
3.6	Analisis dan Evaluasi Hasil Eksperimen . . . . .	33
<b>IV</b>	<b>HASIL DAN PEMBAHASAN . . . . .</b>	<b>35</b>
4.1	Pembangunan <i>Virtual Machine</i> (VM) di DigitalOcean . . . . .	35
4.2	Pemasangan dan Konfigurasi Perangkat Lunak . . . . .	35
4.3	Eksperimen . . . . .	38
4.4	Data Keluaran yang Dihasilkan . . . . .	38
4.5	Analisis dan Evaluasi Hasil Eksperimen: Kinerja . . . . .	41
4.5.1	Persebaran Waktu Eksekusi pada Hadoop dan Spark . . . . .	41
4.5.2	Persebaran <i>Throughput</i> pada Hadoop dan Spark . . . . .	42
4.5.3	Rata-rata Waktu Eksekusi pada Hadoop dan Spark . . . . .	44
4.5.4	Rata-rata <i>Throughput</i> pada Hadoop dan Spark . . . . .	46
4.5.5	Laju Perubahan ( <i>Rate of Change</i> ) . . . . .	48
4.6	Analisis dan Evaluasi Hasil Eksperimen: Penggunaan Sumber Daya . . . . .	58
4.6.1	Penggunaan CPU . . . . .	58
4.6.2	Utilisasi Sistem . . . . .	65
4.7	Perbandingan dengan Penelitian Sebelumnya . . . . .	68

<b>V PENUTUP</b>	<b>70</b>
5.1 Kesimpulan	70
5.2 Saran	70
<b>DAFTAR PUSTAKA</b>	<b>71</b>
<b>LAMPIRAN</b>	<b>76</b>
A Pembuatan <i>Virtual Machine</i> (VM) pada DigitalOcean	77
B Instalasi dan Konfigurasi Perangkat Lunak Prasyarat	80
C Instalasi dan Konfigurasi Hadoop	83
D Instalasi dan Konfigurasi Spark	88
E Instalasi dan Konfigurasi HiBench	89
F Skrip Otomatisasi Eksperimen	92
G Visualisasi Utilisasi Sistem Sesuai Input Data ( <i>Sort</i> )	95
H Visualisasi Utilisasi Sistem Sesuai Input Data ( <i>Word Count</i> )	100

## DAFTAR GAMBAR

Gambar 2.1	Contoh Shell Script yang Digunakan pada Penelitian . . . . .	8
Gambar 2.2	Cara Kerja MapReduce . . . . .	9
Gambar 2.3	Arsitektur Hadoop . . . . .	10
Gambar 2.4	Mode Kerja Hadoop [36] . . . . .	11
Gambar 2.5	Arsitektur HDFS [38] . . . . .	11
Gambar 2.6	Arsitektur YARN [40] . . . . .	12
Gambar 2.7	Komponen Spark . . . . .	13
Gambar 2.8	Arsitektur Spark . . . . .	14
Gambar 2.9	Integrasi Spark dan Hadoop . . . . .	14
Gambar 2.10	<i>Data Sharing</i> pada MapReduce [43] . . . . .	15
Gambar 2.11	<i>Data Sharing</i> pada RDD [43] . . . . .	16
Gambar 2.12	Proses yang Terjadi di HiBench [18] . . . . .	17
Gambar 2.13	Contoh Input dan Output <i>Word Count</i> . . . . .	20
Gambar 2.14	Contoh Input dan Output <i>Sort</i> . . . . .	20
Gambar 2.15	Implementasi MapReduce pada Word Count [45] . . . . .	21
Gambar 2.16	Data Keluaran HiBench dan Dool . . . . .	22
Gambar 3.1	Diagram Alir Penelitian . . . . .	23
Gambar 3.2	Alur Instalasi Perangkat Lunak . . . . .	26
Gambar 3.3	Alur Instalasi Perangkat Lunak Prasyarat . . . . .	27
Gambar 3.4	Alur Instalasi dan Konfigurasi Hadoop . . . . .	28
Gambar 3.5	Alur Instalasi dan Konfigurasi Spark . . . . .	29
Gambar 3.6	Alur Instalasi dan Konfigurasi HiBench . . . . .	29
Gambar 3.7	Total Percobaan . . . . .	30
Gambar 3.8	<i>End-to-end</i> Penelitian . . . . .	31
Gambar 3.9	Contoh Percobaan . . . . .	32
Gambar 4.1	Tampilan Dasbor VM DigitalOcean . . . . .	35
Gambar 4.2	Pengecekan Versi Hadoop . . . . .	36
Gambar 4.3	Pengecekan Versi Spark . . . . .	36
Gambar 4.4	Pengecekan <i>Service</i> yang Berjalan (Normal) . . . . .	36
Gambar 4.5	Pengecekan <i>Service</i> yang Berjalan (Hadoop) . . . . .	37
Gambar 4.6	Pengecekan <i>Service</i> yang Berjalan (Spark) . . . . .	37
Gambar 4.7	Data HiBench <i>Report</i> . . . . .	39
Gambar 4.8	Berkas Dool . . . . .	39
Gambar 4.9	Contoh Data Dool . . . . .	40

Gambar 4.10 Persebaran Waktu Eksekusi <i>Sort</i> (Hadoop, Spark) . . . . .	41
Gambar 4.11 Persebaran Waktu Eksekusi <i>Word Count</i> (Hadoop, Spark) . .	42
Gambar 4.12 <i>Throughput Sort</i> (Hadoop, Spark) . . . . .	43
Gambar 4.13 <i>Throughput Word Count</i> (Hadoop, Spark) . . . . .	44
Gambar 4.14 Rata-rata Waktu Eksekusi ( <i>Sort</i> ) . . . . .	45
Gambar 4.15 Rata-rata Waktu Eksekusi ( <i>Word Count</i> ) . . . . .	45
Gambar 4.16 Rata-rata <i>Throughput</i> ( <i>Sort</i> ) . . . . .	46
Gambar 4.17 Rata-rata <i>Throughput</i> ( <i>Word Count</i> ) . . . . .	47
Gambar 4.18 Laju Perubahan Waktu Eksekusi terhadap Input Data ( <i>Sort</i> )	53
Gambar 4.19 Laju Perubahan Waktu Eksekusi terhadap Input Data ( <i>Word Count</i> ) . . . . .	54
Gambar 4.20 Laju Perubahan <i>Throughput</i> terhadap Input Data ( <i>Sort</i> . . .	55
Gambar 4.21 Laju Perubahan <i>Throughput</i> terhadap Input Data ( <i>Word Count</i> .	56
Gambar 4.22 Laju Perubahan Waktu Eksekusi Hadoop-Spark terhadap Input Data . . . . .	57
Gambar 4.23 Penggunaan CPU ( <i>Sort</i> ) . . . . .	60
Gambar 4.24 Penggunaan CPU ( <i>Word Count</i> ) . . . . .	61
Gambar 4.25 Perbandingan <i>State</i> ( <i>Sort</i> ) . . . . .	63
Gambar 4.26 Perbandingan <i>State</i> ( <i>Word Count</i> ) . . . . .	64
Gambar 4.27 Utilisasi Sistem ( <i>Sort</i> ) pada Input Data 100 KB . . . . .	65
Gambar 4.28 Utilisasi Sistem ( <i>Sort</i> ) pada Input Data 15 GB . . . . .	66
Gambar 4.29 Utilisasi Sistem ( <i>Word Count</i> ) pada Input Data 100 KB . . .	67
Gambar 4.30 Utilisasi Sistem ( <i>Word Count</i> ) pada Input Data 15 GB . . .	67
Gambar 4.31 Rasio Peningkatan Performa Spark-Hadoop Berdasarkan Input Data [1] . . . . .	68

## **DAFTAR TABEL**

Tabel 2.1	Penelitian Terdahulu . . . . .	4
Tabel 2.2	Lanjutan Penelitian Terdahulu . . . . .	5
Tabel 2.3	Beban Kerja pada HiBench [18] . . . . .	18
Tabel 3.1	Konfigurasi Perangkat Keras . . . . .	25
Tabel 3.2	Perangkat Lunak yang Dibutuhkan . . . . .	25
Tabel 3.3	Variasi Input Data . . . . .	32
Tabel 4.1	Konfigurasi HiBench . . . . .	38
Tabel 4.2	Konfigurasi Spark . . . . .	38
Tabel 4.3	Rasio Peningkatan Performa Spark-Hadoop Berdasarkan Input Data (Baru) . . . . .	69

# BAB I

## PENDAHULUAN

### 1.1 Latar Belakang

Perusahaan dan organisasi menghasilkan dan menyimpan data dalam skala besar setiap hari dengan tingkat pertumbuhannya yang dinamis [1]. Pertumbuhan jumlah data diperkirakan akan meningkat hingga 5x lipat pada tahun 2025 dengan *Global Datasphere* diproyeksikan tumbuh dari 33 *Zettabytes* (ZB) pada tahun 2018 menjadi 175 ZB pada tahun 2025 [2]. Jumlah data tersebut membutuhkan pengolahan dengan kecepatan tinggi sehingga dapat dimanfaatkan untuk keperluan bisnis dan pengambilan keputusan [3]. Sebagai contoh, analisis data transaksi nasabah pada perbankan dapat digunakan untuk mendeteksi kecurangan dan meningkatkan keamanan [4], data pasien pada bidang kesehatan dapat memantau wabah penyakit dan menemukan pola pengobatan yang optimal [5], dan data interaksi pengguna pada *e-commerce* diolah untuk memberikan rekomendasi produk personal dan merancang strategi peningkatan penjualan [6].

Data interaksi pengguna pada bidang *e-commerce* dapat dianalisis dengan menerapkan algoritma *sort* dan *word count*, seperti *term frequency-inverse document frequency* (TF-IDF) dan *bag-of-words* (BoW) dimana akan menganalisis kata kunci yang sering muncul dalam deskripsi produk yang dilihat oleh pengguna, sistem dapat merekomendasikan produk serupa atau bahkan produk yang lebih relevan dengan preferensi pengguna tersebut [7].

Semakin besar data yang bisa ditangani, semakin banyak peluang analisis dan nilai yang bisa dihasilkan. Namun, semakin besar volume data yang harus diolah, semakin kompleks pula tantangan yang dihadapi dalam mengelolanya secara efisien dan efektif [8]. Tantangan utama dalam mengelola volume data yang besar adalah memastikan ketersediaan sumber daya komputasi yang memadai. Pendekatan konvensional pemrosesan data besar seperti menambah kapasitas penyimpanan pada perangkat komputasi yang sama dan penggunaan sistem basis data *not only SQL* (NoSQL) memungkinkan pengolahan data menjadi *scalable* dan fleksibel. Namun, ketika skala dan kompleksitas data semakin meningkat, komputasi terdistribusi menjadi pilihan yang lebih tepat karena memiliki sifat *fault-tolerant*, yaitu kemampuan agar tetap beroperasi normal walaupun mengalami kegagalan sebagian dari sistemnya tersebut [9].

Pengolahan data besar yang melibatkan penerapan algoritma seperti *sort* dan *word count* apabila dijalankan secara serial akan memakan waktu pemrosesan yang

lama, terutama dengan volume data yang besar. Oleh karena itu, solusi komputasi terdistribusi diperlukan untuk meningkatkan efisiensi dan kecepatan pemrosesan data. Dua teknologi yang umum digunakan dalam komputasi terdistribusi adalah Hadoop dan Spark [10]. Hadoop dan Spark adalah *platform* komputasi *big data* yang populer dan banyak digunakan di seluruh dunia. Pada tahun 2027, Hadoop diprediksi akan memiliki peningkatan *market size* sebesar USD 341.4 miliar, dari USD 35.3 miliar pada tahun 2024 [11].

Hadoop dan Spark menawarkan berbagai kemampuan untuk mengelola, menyimpan, dan menganalisis data dalam skala besar. Hadoop dan Spark sama-sama memanfaatkan teknik MapReduce untuk memproses data secara terdistribusi [12]. MapReduce merupakan pendekatan yang efektif dalam komputasi terdistribusi karena memungkinkan pemrosesan data yang besar dan kompleks dengan membagi tugas menjadi dua tahap utama, yaitu *map* dan *reduce*. Meskipun sama-sama menggunakan teknik MapReduce, Hadoop dan Spark memiliki skema implementasi yang berbeda. Hadoop menggunakan *Hadoop Distributed File System* (HDFS) sebagai sistem penyimpanan data [13], sementara Spark menggunakan *Resilient Distributed Datasets* (RDDs) yang bersifat *in-memory*. HDFS merupakan sistem penyimpanan berkas terdistribusi yang dirancang untuk menangani data dalam skala besar, sedangkan RDDs memungkinkan penyimpanan data secara sementara di memori, sehingga memungkinkan Spark untuk memproses data lebih cepat [14].

Perbedaan skema tersebut memungkinkan terdapat perbedaan performansi yang dihasilkan dari data yang diproses. Salah satu cara untuk membandingkan performa keduanya adalah menggunakan HiBench. HiBench adalah alat ukur yang dirancang khusus untuk mengukur kinerja sistem *big data*, termasuk Hadoop dan Spark [15]. HiBench memiliki *benchmark* yang dirancang khusus untuk mengukur kinerja suatu algoritma seperti *word count* dan *sort* yang diimplementasikan pada sistem rekomendasi. HiBench juga dapat digunakan pada berbagai *platform cloud*, termasuk DigitalOcean.

Penelitian ini bertujuan untuk mengevaluasi dan membandingkan kinerja Hadoop dan Spark pada *platform cloud* DigitalOcean menggunakan HiBench. Fokus penelitian ini adalah pada beban kerja *word count* dan *sort*, yang mewakili tugas pemrosesan data yang umum. Dengan menganalisis waktu eksekusi, *throughput*, dan penggunaan sumber daya (CPU, memori, I/O), penelitian ini bertujuan untuk memberikan wawasan tentang kekuatan dan kelemahan relatif dari Hadoop dan Spark dalam pemrosesan data. Temuan penelitian ini akan memberikan panduan bagi para praktisi dan peneliti di bidang *big data* untuk memilih *platform* yang tepat untuk kebutuhan pemrosesan data mereka.

## **1.2 Rumusan Masalah**

Adapun rumusan masalah dalam penelitian ini adalah sebagai berikut:

1. Bagaimana performa Hadoop dan Spark dalam hal waktu eksekusi dan *throughput* saat menjalankan beban kerja *word count* dan *sort* pada *platform cloud* DigitalOcean?
2. Bagaimana pola penggunaan performa sumber daya (CPU, memori, I/O) oleh Hadoop dan Spark saat menjalankan beban kerja *word count* dan *sort* dengan berbagai ukuran data?

## **1.3 Tujuan**

Penelitian ini memiliki tujuan, yaitu:

1. Menganalisis dan membandingkan performa Hadoop dan Spark dalam hal waktu eksekusi dan *throughput* saat menjalankan beban kerja *word count* dan *sort* pada *platform cloud* DigitalOcean.
2. Mengidentifikasi pola performa penggunaan sumber daya (CPU, memori, I/O) oleh Hadoop dan Spark saat menjalankan beban kerja *word count* dan *sort* dengan berbagai ukuran data.

## **1.4 Batasan Masalah**

Penelitian ini memiliki beberapa batasan yang perlu diperhatikan sebagai berikut:

1. Performa pada penelitian ini berdasarkan pada waktu eksekusi, *throughput*, dan penggunaan sumber daya (CPU, memori, I/O).
2. Penelitian ini akan fokus pada perbandingan kinerja antara Hadoop dan Spark dalam mode *pseudo-distributed* dengan input data berupa teks.
3. Pengukuran performa akan menggunakan HiBench (tolok ukur utama) dan Dool *Monitoring System* (tolok ukur pembantu)
4. Implementasi Hadoop dan Spark akan menggunakan salah satu penyedia layanan awan, yaitu *DigitalOcean*.

## BAB II

### LANDASAN TEORI

#### 2.1 Tinjauan Pustaka

Penelitian ini menggunakan beberapa teori dasar supaya memperjelas proses penelitian dan memberikan pemahaman lebih lanjut. Peneltian terdahulu mengenai evaluasi performa Hadoop dan Spark dapat dilihat pada Tabel 2.1.

**Tabel 2.1** Penelitian Terdahulu

Tahun	Judul	Penulis	Metode	Deskripsi Penelitian
2020	<i>A comprehensive performance analysis of Apache Hadoop and Apache Spark for large scale data sets using HiBench [16]</i>	N. Ahmed, Andre L. C. Barczak, Teo Susnjak, Mohammed A. Rashid	Penelitian ini menyelidiki parameter-parameter yang paling berdampak, yaitu <i>input splits</i> , dan <i>shuffle</i> , untuk membandingkan kinerja antara Hadoop dan Spark, dengan menggunakan klaster yang diimplementasikan di laboratorium. Guna mengevaluasi kinerja, dua beban kerja dipilih, yakni <i>WordCount</i> dan <i>TeraSort</i> . Metrik kinerja diukur berdasarkan tiga kriteria: waktu eksekusi, <i>throughput</i> , dan <i>speedup</i> .	Kinerja kedua sistem sangat bergantung pada ukuran data masukan dan pemilihan parameter yang tepat. Analisis hasil menunjukkan bahwa Spark memiliki kinerja yang lebih baik dibandingkan dengan Hadoop ketika set data kecil, mencapai peningkatan kecepatan hingga dua kali lipat dalam beban kerja <i>WordCount</i> dan hingga 14 kali lipat dalam beban kerja <i>TeraSort</i> ketika nilai parameter <i>default</i> dikonfigurasi ulang. Hadoop memerlukan waktu yang lebih sedikit dibandingkan dengan Spark. Hal tersebut karena nilai <i>throughput</i> dan <i>throughput/node</i> Hadoop lebih tinggi daripada Apache Spark.
2020	Perbandingan Kinerja Komputasi Hadoop dan Spark untuk Memprediksi Cuaca (Studi Kasus: <i>Storm Event Database</i> [10])	Rendiyono Wahyu Saputro, Aminuddin, Yuda Munarko	Mengimplementasikan gugus komputer untuk memproses dataset dengan berbagai ukuran dan dalam jumlah komputer yang berbeda.	
2018	<i>Performance comparison between Hadoop and Spark frameworks using HiBench benchmarks [11]</i>	Yassir Samadi, Mostapha Zbak, Claude Tadonki	Perbandingan kinerja diimplementasikan pada mesin virtual (VM). Untuk membandingkannya, digunakan HiBench. Perbandingan dilakukan berdasarkan tiga kriteria: waktu eksekusi, <i>throughput</i> , dan <i>speedup</i> . Beban kerja <i>WordCount</i> diuji dengan ukuran data yang berbeda.	Spark lebih efisien dibandingkan Hadoop dalam menangani jumlah data yang besar. Namun, Spark memerlukan alokasi memori yang lebih tinggi, karena memuat data yang akan diproses ke dalam memori dan menyimpannya dalam cache untuk sementara.

**Tabel 2.2** Lanjutan Penelitian Terdahulu

Tahun	Judul	Penulis	Metode	Deskripsi Penelitian
2015	<i>Comparing Apache Spark and Map Reduce with Performance Analysis using K-Means [17]</i>	Satish Gopalani, Rohan Arora	Hadoop dan Spark dibandingkan menggunakan algoritma pembelajaran mesin (KMeans). Ukuran data yang digunakan adalah sebesar 64MB, 1240MB dengan satu node, dan 1240MB dengan dua node.	Hasil-hasil penelitian dengan jelas menunjukkan bahwa kinerja Spark jauh lebih tinggi dari segi waktu, di mana setiap ukuran dataset mengakibatkan penurunan waktu pemrosesan hingga tiga kali lipat dibandingkan dengan Hadoop.

## 2.2 Konsep *Big Data*

*Big Data* biasanya sering didefinisikan bersama dengan kompleksitas suatu data [18]. Berbeda dengan tradisional data, *Big Data* merujuk pada pertumbuhan data dalam berbagai format, baik dari struktur, semi-terstruktur, dan tidak terstruktur [19]. *Big Data* memiliki banyak jenis sehingga membutuhkan teknologi yang lebih bertenaga serta algoritma yang lebih canggih. Pendekatan teknologi yang sering digunakan oleh *Business Intelligence* biasanya tidak dapat lagi efisien jika digunakan. *Big Data* biasanya didefinisikan menjadi tiga karakteristik (3V), yaitu *Volume*, *Velocity*, dan *Variety* [20]. *Volume* berkaitan dengan jumlah data yang terbentuk atau dibuat secara terus menerus oleh beragam perangkat, seperti telepon genggam dan aplikasi (sosial media, sensor, IoT). Jumlah data diharapkan tumbuh 5x lipat pada tahun 2020 [20]. Selanjutnya, *Velocity* memberikan makna bahwa data bertumbuh secara cepat dan harus diproses secara cepat juga untuk memberikan informasi yang berguna [21]. YouTube adalah ilustrasi yang tepat untuk menggambarkan bagaimana pertumbuhan data begitu cepat. Terakhir, *Variety* berkaitan dengan variasi sumber dan format data.

Penerapan dari *big data* tidak hanya terbatas pada pengumpulan dan penyimpanan data, tetapi juga meliputi analisis dan pengolahan data tersebut untuk menghasilkan wawasan yang berguna. Beberapa sektor yang telah menerapkan *big data* secara luas meliputi kesehatan, keuangan, ritel, dan pemerintahan [19]. Dalam sektor kesehatan, *big data* digunakan untuk menganalisis informasi pasien secara massal guna meningkatkan kualitas perawatan dan menemukan pola-pola penyakit. Se-mentara itu, di sektor keuangan, *big data* membantu dalam analisis risiko, deteksi penipuan, dan personalisasi layanan untuk pelanggan.

## 2.3 Ekstraksi Fitur Teks (*Text Feature Extraction*)

*Ekstraksi Fitur Teks* adalah salah satu proses pada pembelajaran mesin (*machine learning*) dan data analisis yang melibatkan identifikasi dan ekstraksi fitur yang re-

levan dari data mentah [22]. Fitur-fitur tersebut nantinya akan digunakan untuk membuat data yang lebih informatif, sehingga dapat digunakan untuk klasifikasi, prediksi, dan klasterisasi. Ekstraksi fitur bertujuan untuk mengurangi kompleksitas data (atau yang sering disebut juga *Data Dimensionality*) namun tetap menyimpan sebanyak mungkin informasi yang paling relevan. Hal ini bertujuan untuk meningkatkan performa dan efisiensi algoritma pada pembelajaran mesin dan mempermudah dalam proses analisis. Ekstraksi fitur dapat melibatkan membuat fitur baru (*Feature Engineering*) atau memanipulasi data yang menghasilkan fitur yang berguna. Ekstraksi fitur juga memainkan peran penting dalam banyak penerapan di dunia nyata, misalnya untuk pemrosesan teks dan *Natural Language Processing* (NLP). Dalam skenario ini, data mentah mungkin mengandung banyak fitur yang tidak relevan atau berlebihan. Hal ini menyulitkan algoritma untuk memproses data secara akurat. Dengan melakukan ekstraksi fitur, fitur yang relevan dipisahkan dari fitur yang tidak relevan [23]. Dengan lebih sedikit fitur yang harus diproses, kumpulan data menjadi lebih sederhana dan akurasi serta efisiensi analisis meningkat.

### 2.3.1 *Bag of Words (BoW)*

BoW adalah teknik sederhana yang mengabaikan urutan dan struktur gramatiskal kalimat, dan hanya berfokus pada frekuensi kemunculan kata dalam dokumen. Prosesnya melibatkan langkah-langkah berikut:

1. **Tokenisasi:** Teks dipecah menjadi kata-kata individual (token).
2. **Pembuatan Kosakata:** Daftar unik dari semua token yang ada dalam seluruh kumpulan dokumen dibuat. Ini disebut "kosakata".
3. **Penghitungan Kata (Word Count):** Untuk setiap dokumen, frekuensi kemunculan setiap kata dalam kosakata dihitung.
4. **Representasi Vektor:** Setiap dokumen diwakili sebagai vektor, di mana setiap elemen vektor mewakili frekuensi kata tertentu dalam kosakata.

BoW mudah diimplementasikan dan efisien, namun kelemahannya adalah kehilangan informasi kontekstual dan semantik. Kata-kata dengan frekuensi tinggi, meskipun kurang informatif, dapat mendominasi representasi vektor.

### 2.3.2 *Term Frequency-Inverse Document Frequency (TF-IDF)*

TF-IDF mengatasi beberapa kelemahan BoW dengan mempertimbangkan pentingnya kata dalam dokumen dan koleksi dokumen [24]. TF-IDF terdiri dari dua komponen:

1. **Term Frequency (TF):** Mengukur seberapa sering suatu kata muncul dalam dokumen.

2. **Inverse Document Frequency (IDF):** Mengukur seberapa penting suatu kata dalam seluruh koleksi dokumen. Kata-kata yang muncul di banyak dokumen (seperti kata "yang") memiliki IDF rendah, sementara kata-kata yang jarang muncul (dan kemungkinan lebih informatif) memiliki IDF tinggi.

Dengan mengalikan TF dan IDF, kita mendapatkan nilai TF-IDF yang mencerminkan pentingnya kata dalam dokumen dan koleksi dokumen.

### 2.3.3 Penggunaan *Word Count* dan *Sort* pada BoW dan TF-IDF

1. **Word Count:** Digunakan dalam kedua metode (BoW dan TF-IDF) untuk menghitung frekuensi kemunculan kata dalam dokumen.
2. **Sort:** Biasanya tidak digunakan secara langsung dalam proses BoW atau TF-IDF. Namun, pengurutan dapat digunakan untuk:
  - **Memilih fitur:** Memilih fitur dengan nilai TF-IDF tertinggi untuk mengurangi dimensi data dan fokus pada kata-kata yang paling informatif.
  - **Visualisasi:** Mengurutkan kata berdasarkan frekuensi atau nilai TF-IDF dapat membantu dalam visualisasi dan analisis data.

## 2.4 Komputasi Awan (*Cloud Computing*)

Komputasi awan didefinisikan sebagai sistem informasi yang memungkinkan akses mudah ke sumber daya komputasi atau layanan komputasi sesuai permintaan (*on demand*), misalnya segala sesuatu mulai dari aplikasi (Google Mail, Microsoft One Drive, Siakad Itera) hingga pusat data di seluruh internet dengan sistem bayar sesuai penggunaan. Sistem komputasi awan saat ini menyediakan tiga layanan utama:

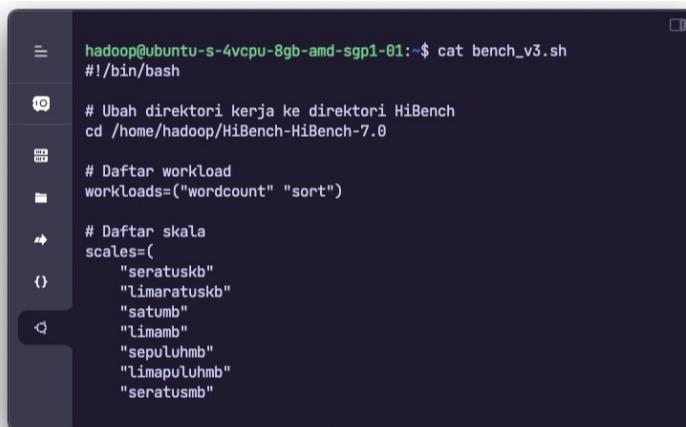
1. *Infrastructure as a service* (IaaS), adalah layanan awan yang menawarkan kepada pengguna untuk mengatur dan mengonfigurasikan sumber daya yang dibutuhkan untuk menjalankan aplikasi dan sistem IT. Jenis IaaS biasanya berbentuk komputasi, penyimpanan, dan sumber daya jaringan yang dibuat sebagai layanan.
2. *Platform as a service* (PaaS), adalah layanan awan yang memungkinkan pengguna untuk mengembangkan, mengelola, dan menjalankan aplikasi di lingkungan yang dikontrol oleh penyedia layanan, tanpa harus khawatir dengan infrastruktur yang mendasarinya.
3. *Software as a service* (SaaS), adalah layanan awan yang mengacu pada aplikasi yang berjalan pada infrastruktur awan yang di-hosting oleh vendor atau penyedia layanan dan tersedia untuk pengguna akhir melalui browser web.

Komputasi awan menjadi salah satu aspek terpenting dalam menjalankan komputasi yang kompleks, misalnya untuk menjalankan Hadoop atau Spark. Salah satu kom-

putasi awan yang dapat diandalkan adalah DigitalOcean. DigitalOcean dibentuk pada tahun 2012 untuk memenuhi kebutuhan pengembang untuk mendapatkan akses komputasi awan yang mudah dimengerti dan terjangkau [25]. Salah satu produk DigitalOcean yang sering digunakan adalah Droplet, *easy-to-use* komputer virtual yang siap digunakan dalam hitungan menit. Pengguna dapat memilih lokasi dimana komputer akan dijalankan, bagaimana konfigurasi prosesor serta memori, memilih sistem operasi apa yang akan digunakan, dan banyak hal lainnya.

## 2.5 *Shell Script*

*Shell script* merupakan serangkaian perintah yang dieksekusi dalam lingkungan sistem operasi Unix atau Unix-like [26]. *Shell script* memungkinkan pengguna untuk mengotomatiskan tugas-tugas rutin, melakukan pemrosesan file, dan bahkan membangun aplikasi yang kompleks dengan menggunakan perintah-perintah shell. *Shell script* umumnya ditulis menggunakan bahasa pemrograman shell, seperti Bash (Bo urne Again Shell), yang merupakan shell standar pada sebagian besar sistem operasi Linux dan MacOS. Sebagai contoh, dalam mengelola pencadangan sistem, seorang administrator dapat membuat *shell script* sederhana yang menjalankan perintah-perintah untuk menyalin file-file penting ke lokasi penyimpanan cadangan secara berkala. Skrip ini dapat dijadwalkan untuk berjalan secara otomatis menggunakan *cron job*, sehingga proses pencadangan dapat dilakukan tanpa campur tangan manusia secara berkala. Dengan menggunakan variabel dan logika sederhana, administrator dapat dengan mudah menyesuaikan skrip ini untuk memenuhi kebutuhan pencadangan spesifik sistem mereka. Dengan demikian, *shell script* tidak hanya menghemat waktu dan tenaga, tetapi juga meningkatkan kehandalan dan konsistensi dalam administrasi sistem.



```
hadoop@ubuntu-s-4vcpu-8gb-amd-sgp1-01:~$ cat bench_v3.sh
#!/bin/bash

# Ubah direktori kerja ke direktori HiBench
cd /home/hadoop/HiBench-HiBench-7.0

# Daftar workload
workloads=("wordcount" "sort")

# Daftar skala
scales=(
    "seratuskb"
    "limaratuskb"
    "satumb"
    "limamb"
    "sepuluhmb"
    "limapuluhmb"
    "seratusmb"
```

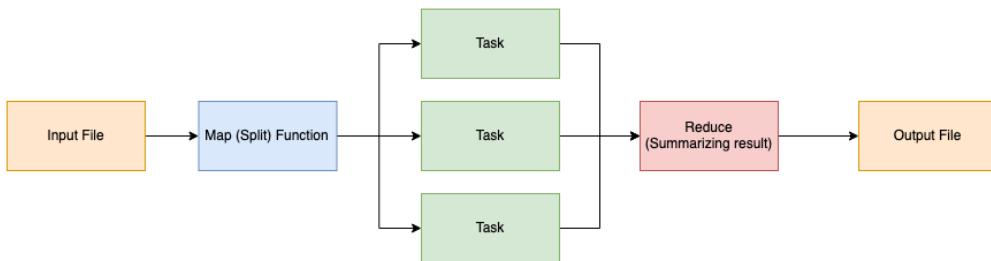
Gambar 2.1 Contoh Shell Script yang Digunakan pada Penelitian

Gambar 2.1 menampilkan sebuah contoh potongan *shell script* yang digunakan dalam penelitian ini. Berikut adalah penjelasan lebih rinci mengenai skrip tersebut,

1. **Baris 1:** Mendefinisikan interpreter Bash untuk menjalankan skrip.
2. **Baris 3-4:** Mengubah direktori kerja ke direktori HiBench.
3. **Baris 6-7:** Mendefinisikan daftar (*list*) *workLoads* berisi macam-macam beban kerja, yaitu *wordcount* dan *sort*.
4. **Baris 9-15:** Mendefinisikan daftar *scales* berisi skala input data.

## 2.6 MapReduce

MapReduce adalah model pemrograman dan implementasi teknik pemrosesan data berukuran besar yang pertama kali dipopulerkan oleh Google pada tahun 2004[27]. MapReduce menawarkan pemrosesan data yang dapat diandalkan serta *fault-tolerant manner* (tahan terhadap kesalahan). MapReduce berjalan secara paralel dan bera-ada pada lingkungan komputasi terdistribusi [28]. Model ini mengadopsi arsitektur tersentraliasi, yaitu satu *node* berperan sebagai *master* dan *node* yang lain berperan sebagai *workers* atau *slave* [29], [30]. *Master node* bertanggung jawab untuk melakukan penjadwalan kerja, dan *slave node* berperan untuk menjalankan eksekusi kerja.

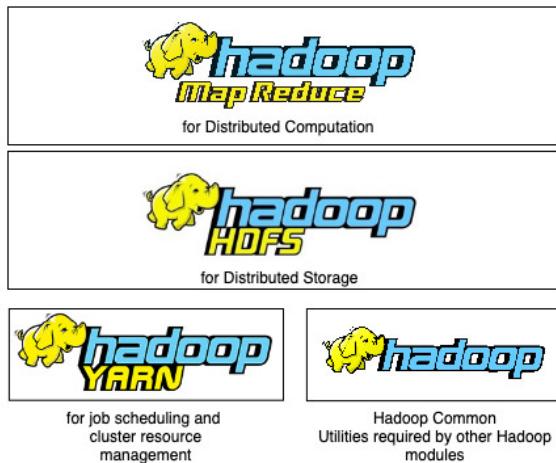


**Gambar 2.2** Cara Kerja MapReduce

MapReduce terdiri dari fungsi *Map* dan fungsi *Reduce* [31]. Kedua fungsi ini tersebar di seluruh *slave node* yang terhubung dalam klaster dan berjalan secara paralel. Fungsi *Map* berperan untuk membagi masalah besar menjadi masalah yang lebih kecil dan mendistribusikannya ke *slave node*. Hasil pemrosesan dari *slave node* akan dikumpulkan oleh *master node* melalui fungsi *Reduce*. Sesuai dengan Gambar 2.2, hasil dari proses *Reduce* yang akan dikirimkan sebagai hasil akhir proses MapReduce.

### 2.6.1 Apache Hadoop

Apache Hadoop adalah perangkat lunak sumber terbuka yang ditulis dengan bahasa pemrograman Java untuk pemrosesan dan penyimpanan data menggunakan komputasi terdistribusi [32]. Hadoop dapat diinstalasi pada satu *node* komputer, atau ratusan *node* komputer yang digabungkan dalam sebuah klaster [33]. Berkaitan dengan pemrosesan data, Hadoop mengimplementasikan model MapReduce untuk pemrosesan data secara paralel dan cepat. Selain itu, Hadoop menyediakan sistem penyimpanan data terdistribusi yang dikenal sebagai Hadoop Distributed File System (HDFS) untuk akses data, pemrosesan, dan komputasi [34]. Arsitektur Hadoop secara umum dapat dilihat pada Gambar 2.3.

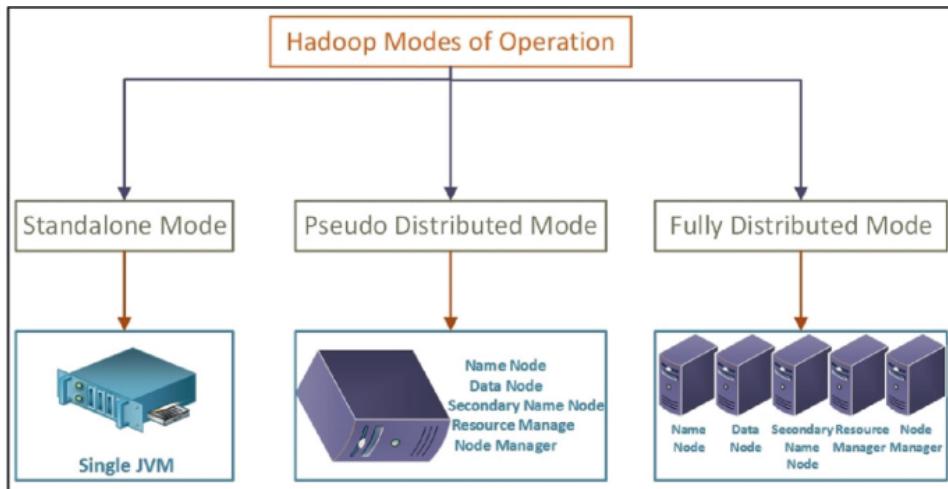


**Gambar 2.3** Arsitektur Hadoop

### 2.6.2 Mode Kerja Hadoop

Hadoop dapat dijalankan dalam tiga mode operasi yang berbeda yaitu *standalone*, *pseudo-distributed*, dan *fully distributed* [35]. Dalam *standalone mode*, semua proses Hadoop berjalan pada satu node tunggal menggunakan sistem berkas lokal tanpa memerlukan konfigurasi kustom pada Hadoop seperti pada Gambar 2.4.

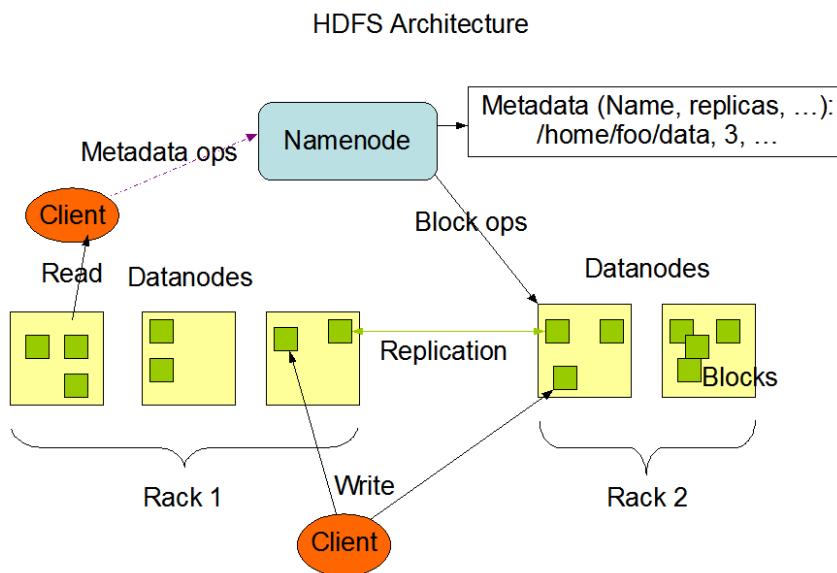
*Pseudo-distributed mode* menjalankan semua komponen Hadoop pada satu node tunggal tetapi menyimulasikan kluster dengan komunikasi antar proses melalui socket jaringan, sehingga memerlukan konfigurasi pada berkas *core-site*, *mapred-site*, dan *hdfs-site*. Sedangkan *fully distributed mode* menyebarkan proses Hadoop ke beberapa node dalam kluster sebenarnya yang biasanya digunakan untuk tahap produksi. *Fully distributed mode* mendukung skalabilitas, ketersediaan tinggi, dan keamanan dengan memerlukan instalasi Hadoop dan konfigurasi kluster pada setiap node.



Gambar 2.4 Mode Kerja Hadoop [36]

### 2.6.3 Hadoop Distributed File System (HDFS)

*Hadoop Distributed File System* adalah sistem file terdistribusi yang dikembangkan sebagai bagian dari Hadoop [37]. HDFS dirancang khusus untuk menyimpan data dalam jumlah besar dan memungkinkan pemrosesan data secara paralel. Beberapa fitur utama dari HDFS antara lain skalabilitas, toleransi kesalahan, *streaming access*, dan cocok untuk aplikasi *batch* seperti MapReduce.



Gambar 2.5 Arsitektur HDFS [38]

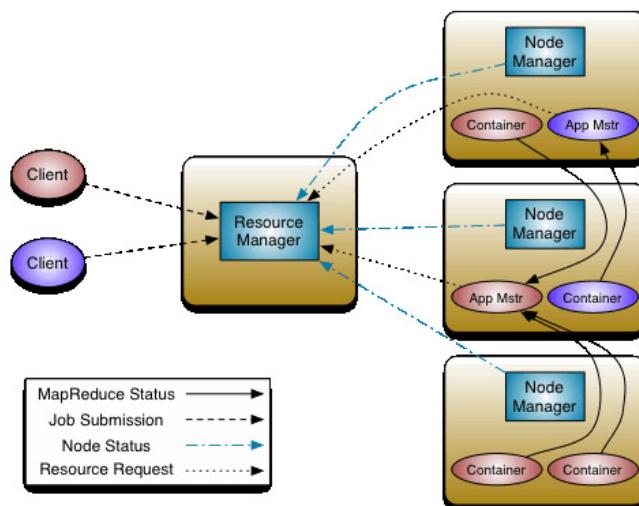
Secara struktur, HDFS terdiri dari NameNode sebagai *node master* yang mengelola *metadata* dan *namespace*, serta DataNode sebagai *node slave* yang bertugas menyimpan data sebenarnya dalam bentuk blok seperti pada Gambar 2.5. Berkas di HDFS dipartisi menjadi satu atau lebih blok berukuran 64MB atau 128MB, kemudian didistribusikan dan disimpan di beberapa DataNode. Setiap blok direplikasi

(biasanya 3x) di DataNode yang berbeda untuk toleransi kesalahan. Replikasi blok di *node/rack* yang berbeda juga meningkatkan ketersediaan HDFS.

Dengan desain terdistribusi, HDFS sangat populer digunakan bersama framework Hadoop untuk memproses *big data* [39]. Namun, ketergantungan pada *single* NameNode dan performa akses data acak yang kurang optimal menjadi kelemahan utama HDFS. Secara keseluruhan, HDFS telah terbukti menjadi pilihan matang untuk penyimpanan data massal secara terdistribusi.

#### 2.6.4 Hadoop YARN

*Hadoop YARN (Yet Another Resource Negotiator)* adalah manajer sumber daya dan sistem penjadwalan untuk kluster Hadoop. Komponen ini diperkenalkan dalam Hadoop 2.x sebagai evolusi dari Hadoop MapReduce 1.x, yang mengintegrasikan manajemen sumber daya dan pemrosesan data dalam satu sistem. YARN memungkinkan kluster untuk menjalankan berbagai aplikasi secara bersamaan dengan efisiensi yang lebih baik. YARN memisahkan fungsi manajemen sumber daya dari mekanisme pemrosesan data, yang sebelumnya keduanya tertanam dalam MapReduce. Dengan demikian, YARN dapat mendukung berbagai paradigma pemrosesan data di atas Hadoop, selain MapReduce, seperti *real-time processing* dan *graph processing*.



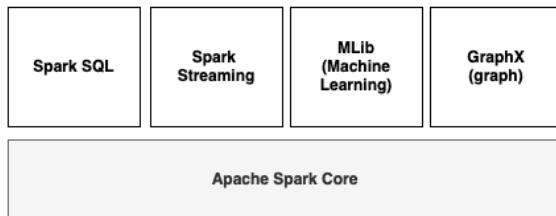
Gambar 2.6 Arsitektur YARN [40]

Struktur utama YARN terdiri dari ResourceManager yang bertugas mengkoordinasikan alokasi sumber daya di seluruh kluster, dan NodeManager yang berjalan di setiap *node* untuk mengawasi penggunaan sumber daya dan mengelola *container* tempat aplikasi dijalankan. ApplicationMaster adalah komponen khusus untuk setiap aplikasi yang bertanggung jawab untuk negosiasi sumber daya dengan Reso-

urceManager dan bekerja dengan NodeManager untuk menjalankan dan memantau *tasks* seperti pada Gambar 2.6.

## 2.7 Apache Spark

Apache Spark diperkenalkan oleh Apache Software Foundation sebagai *framework* pemrosesan data paralel *open-source* yang dirancang untuk mempercepat pemrosesan *big data* dibandingkan dengan Hadoop MapReduce [41]. Meskipun sama-sama menggunakan model pemrosesan MapReduce, Spark bukanlah hasil modifikasi dari Hadoop MapReduce[8]. Hal ini dikarenakan Spark menggunakan teknologi tersendiri yaitu *Resilient Distributed Datasets* (RDDs) yang memungkinkan Spark memproses data secara *in-memory* sehingga lebih cepat. Selain itu, Spark memiliki klaster pengolahan data tersendiri sehingga dapat berjalan independen tanpa Hadoop. Dengan performa tinggi serta dukungan untuk pemrosesan data secara interaktif, Spark banyak digunakan untuk pemrosesan data skala besar. Komponen yang terdapat pada Spark dapat dilihat pada Gambar 2.7

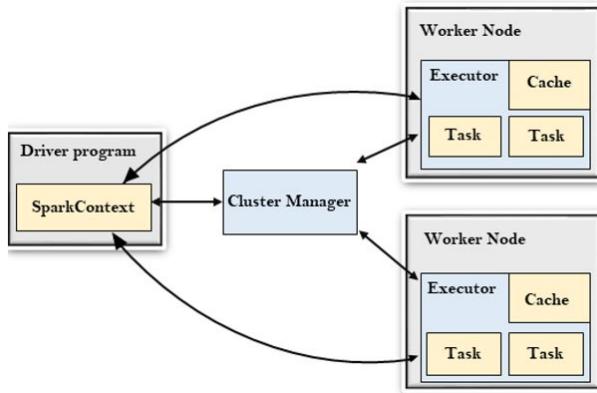


Gambar 2.7 Komponen Spark

### 2.7.1 Arsitektur Spark

Arsitektur Spark dirancang untuk pemrosesan data terdistribusi yang efisien dan cepat seperti pada Gambar 2.8. Komponen utamanya meliputi *Spark Driver*, *Cluster Manager*, dan *Spark Executor*. *Spark Driver* berperan sebagai otak operasi, bertanggung jawab untuk mengonversi program pengguna menjadi tugas-tugas, menjadwalkan tugas pada *executor*, dan mengelola keseluruhan alur kerja. *Cluster Manager*, yang dapat berupa YARN, Mesos, atau mode *standalone* Spark, menangani alokasi sumber daya dan peluncuran *executor* pada *node-node cluster*. *Spark Executor*, yang berjalan pada *node-node cluster*, menjalankan tugas-tugas pemrosesan data yang diberikan oleh *driver* dan menyediakan penyimpanan dalam memori untuk data yang *di-cache*. Interaksi antara komponen-komponen ini memungkinkan pemrosesan data paralel yang cepat dan toleransi kesalahan yang tinggi. Arsitektur Spark yang fleksibel mendukung berbagai bahasa pemrograman dan sistem pe-

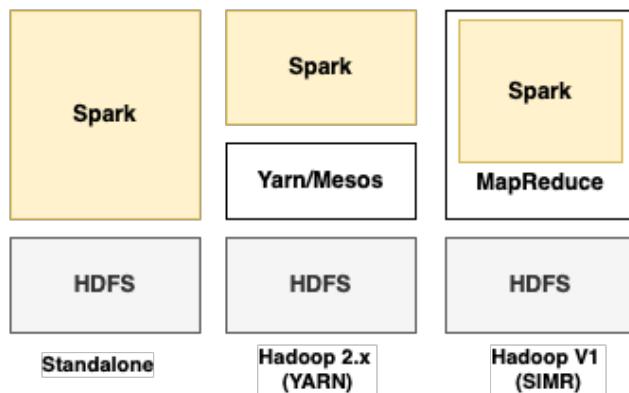
nyimpanan data, menjadikannya solusi ideal untuk berbagai kasus penggunaan data besar.



**Gambar 2.8** Arsitektur Spark

### 2.7.2 Integrasi Hadoop dan Spark

Integrasi Spark dengan Hadoop dapat dilakukan melalui tiga metode berbeda seperti pada Gambar 2.9 [42]. Pertama, metode *Standalone* mengharuskan Spark menempati tempat di atas HDFS (*Hadoop Distributed File System*). Dalam skenario ini, Spark dan MapReduce berjalan berdampingan untuk menangani semua pekerjaan Spark pada kluster. Kedua, metode Hadoop Yarn memungkinkan Spark berjalan pada Yarn tanpa memerlukan instalasi sebelumnya atau akses *root*. Hal ini memfasilitasi integrasi Spark ke dalam ekosistem Hadoop, atau memungkinkan komponen lain berjalan di atas integrasi Hadoop dan Spark. Terakhir, metode *Spark in MapReduce* (SIMR). Dengan SIMR, pengguna dapat memulai Spark dan menggunakan *shell*-nya tanpa memerlukan akses administratif.



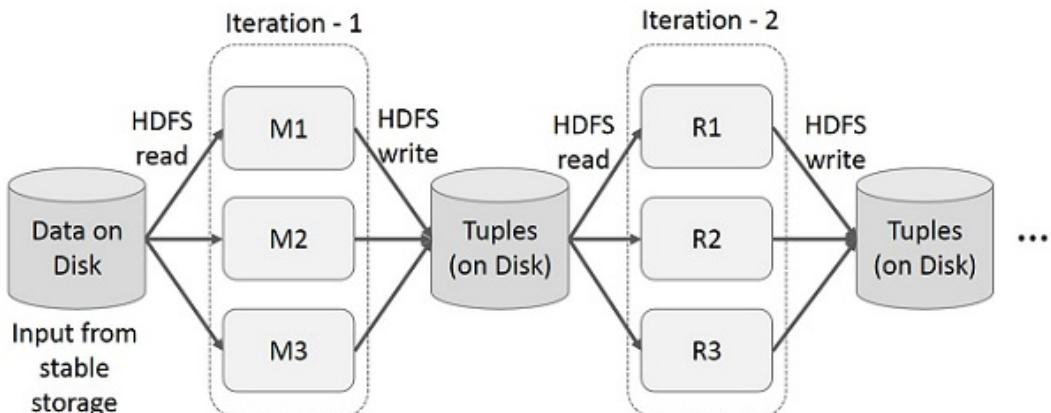
**Gambar 2.9** Integrasi Spark dan Hadoop

### 2.7.3 Keterbatasan *Data Sharing* pada MapReduce

MapReduce, sebagai kerangka kerja pemrosesan data terdistribusi, mengandalkan sistem penyimpanan eksternal yang stabil, seperti HDFS, untuk berbagi data antar tugas (*job*). Hal ini mengakibatkan ineffisiensi karena beberapa alasan, yaitu:

1. **Replikasi Data:** Data perlu direplikasi ke beberapa node untuk toleransi kesalahan dan paralelisme. Replikasi ini memakan waktu dan bandwidth jaringan.
2. **Serialisasi/Deserialisasi:** Data perlu diubah formatnya (serialisasi) sebelum dikirim melalui jaringan dan diubah kembali (deserialisasi) di simpul tujuan. Proses ini menambah beban komputasi.
3. **Disk I/O:** Akses data dari dan ke disk cenderung lambat dibandingkan dengan akses memori. Pada MapReduce, setiap operasi baca-tulis data melibatkan interaksi dengan *disk*, yang memperlambat performa.

Keterbatasan ini terlihat jelas pada aplikasi yang membutuhkan operasi iteratif, di mana hasil antara satu tugas perlu digunakan kembali oleh tugas berikutnya. Pada MapReduce, setiap iterasi memerlukan pembacaan dan penulisan data ke HDFS, seperti yang diilustrasikan pada Gambar 2.10. Akibatnya, aplikasi iteratif pada MapReduce cenderung lambat dan tidak efisien.



Gambar 2.10 *Data Sharing* pada MapReduce [43]

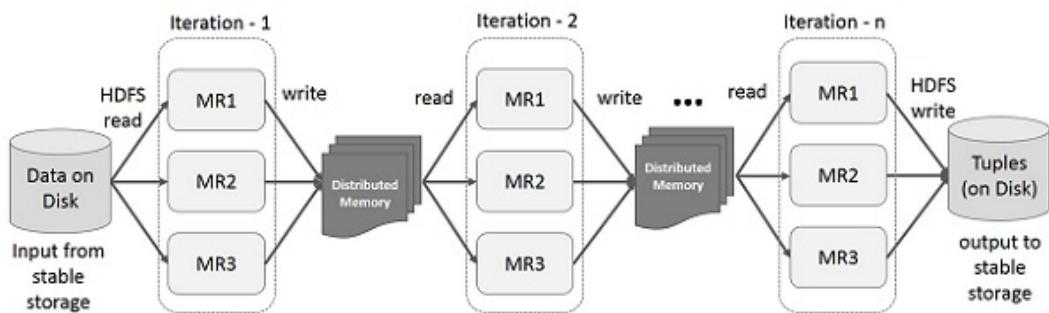
### 2.7.4 Solusi *Data Sharing* dengan Spark RDD

Spark mengatasi keterbatasan MapReduce dengan memperkenalkan RDD, yaitu koleksi data terdistribusi yang disimpan dalam memori. RDD bersifat *immutable*, artinya data tidak dapat diubah setelah dibuat, dan *fault-tolerant*, artinya data dapat dipulihkan jika terjadi kegagalan node.

Dengan menyimpan data dalam memori, RDD memungkinkan akses data yang jauh lebih cepat dibandingkan dengan akses disk pada MapReduce. Selain itu, RDD

mendukung *lazy evaluation*, di mana operasi pada RDD tidak dieksekusi langsung, melainkan disimpan sebagai *lineage* atau urutan operasi yang perlu dilakukan. Hal ini memungkinkan Spark untuk mengoptimalkan eksekusi tugas dan mengurangi overhead komputasi.

Pada aplikasi iteratif, RDD dapat menyimpan hasil antara dalam memori dan membagikannya antar tugas tanpa perlu mengakses disk, seperti yang ditunjukkan pada Gambar 2.11. Dengan demikian, Spark RDD memungkinkan eksekusi aplikasi iteratif yang jauh lebih cepat dan efisien dibandingkan dengan MapReduce.

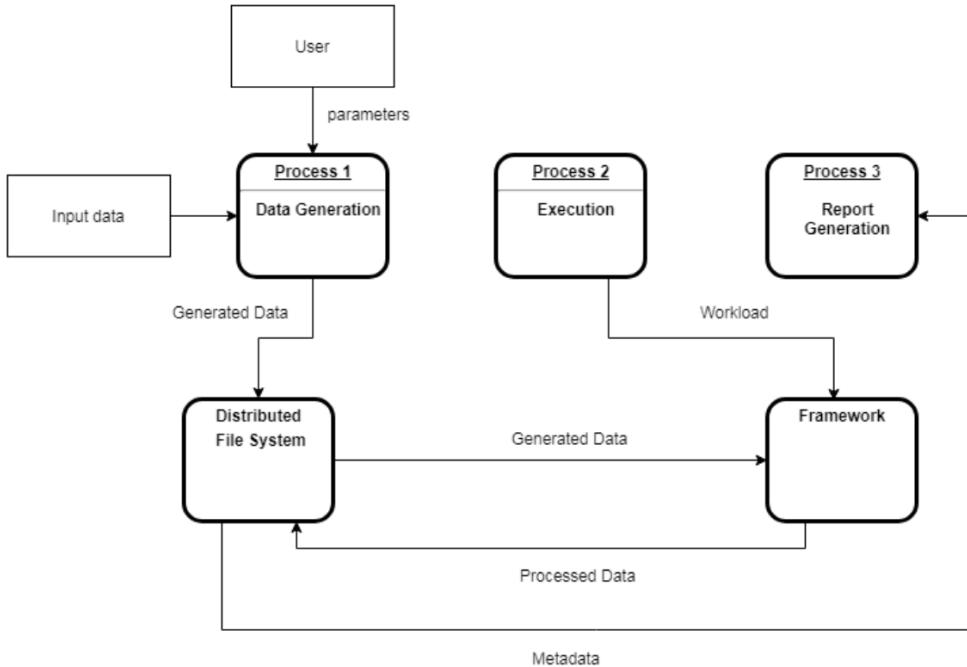


Gambar 2.11 Data Sharing pada RDD [43]

## 2.8 HiBench

HiBench memudahkan dalam eksekusi pengukuran berbagai beban kerja karena HiBench sudah membungkus sekumpulan perintah dalam bentuk *shell script*[1]. Pengguna hanya perlu menjalankan perintah untuk HiBench melakukan persiapan data. Selanjutnya, pengguna bisa langsung melakukan pengukuran beban kerja. Hasilnya dapat terlihat langsung pada laporan HiBench. Secara umum, alur kerja HiBench terlihat seperti pada Gambar 2.12.

HiBench terdiri dari 3 proses utama. Proses pertama, pengguna melakukan konfigurasi parameter *Data Generation*. Selanjutnya, *Data Generation* akan melakukan pembentukan data yang nantinya akan disimpan pada *Distributed File System* (DFS). Data ini yang akan digunakan pada proses selanjutnya. Proses kedua adalah proses eksekusi. Pengguna akan memicu salah satu beban kerja pada HiBench. Selanjutnya, HiBench akan memberi perintah kepada perangkat lunak (Hadoop/Spark) untuk menjalankan beban kerja tersebut. Setiap melakukan pengukuran, data yang digunakan adalah data dari *Distributed File System* yang sebelumnya sudah dibentuk. Hasil dari eksekusi ini akan disimpan kembali di DFS. Proses terakhir adalah proses pembentukan laporan. Hasil dari proses sebelumnya akan diambil serta akan dibuatkan laporan secara otomatis. Dalam laporan otomatis yang diberikan oleh HiBench, terdapat beberapa metriks yang tersedia, meliputi *Execution*



**Gambar 2.12** Proses yang Terjadi di HiBench [18]

*Time* dan *Throughput*. *Execution Time* memiliki makna seberapa lama suatu kejadian berlangsung. Waktu yang dihitung adalah waktu diantara waktu awal dan waktu terakhir kejadian. Metriks ini dihitung dalam skala detik. Selanjutnya, *Throughput* menghitung berapa banyak unit informasi yang dapat diproses oleh sistem dalam waktu tertentu. Metriks ini dinyatakan dalam *byte/detik*.

### 2.8.1 Beban Kerja *Micro Benchmark* dan Sumber Data

HiBench versi 7.1 memiliki 29 beban kerja (*workload*) yang dapat diuji [44]. Beban kerja ini dikategorikan menjadi 7 kategori, yaitu *micro*, *ml* (*machine learning*), *sql*, *graph*, *websearch and streaming*. Tabel 2.3 menunjukkan macam-macam beban kerja yang dapat diuji. *Workload name* mengindikasikan algoritma utama atau operasi apa yang dilakukan. *Workload type* merepresentasikan kategori dari beban kerja. *Operation types* menunjukkan klasifikasi jenis operasi yang dilakukan. *Workload Submission Policy* berguna untuk mengetahui bagaimana cara pengguna untuk mengatur atau mengonfigurasikan beban kerja.

Beban kerja *micro benchmarks* merupakan kategori khusus yang dirancang untuk menguji kemampuan *raw processing power* [18]. Dalam kategori ini, terdapat dua beban kerja populer, yaitu Sort, dan WordCount [15]. Beban kerja Sort dan WordCount merepresentasikan pekerjaan MapReduce [12]. Beban kerja Sort akan mengurutkan setiap kata dalam berkas input. Beban kerja WordCount akan melakukan tugas pemetaan (*map task*) dan mengeluarkan output (kata, 1) untuk setiap

**Tabel 2.3** Beban Kerja pada HiBench [18]

Workload Name	Workload Type	Operation Type	Workload Submission Policy	Software Stack
Sort	Micro Benchmark	Algorithm	Pre-Specified Process	Hadoop, Spark
WordCount	Micro Benchmark	Algorithm	Pre-Specified Process	Hadoop, Spark
Terasort	Micro Benchmark	Algorithm	Pre-Specified Process	Hadoop, Spark
Sleep	Micro Benchmark	Algorithm	Pre-Specified Process	Hadoop, Spark
enhanced DFSIO	Micro Benchmark	IO	Parameter Control	Hadoop, Spark
Bayesian Classification	Machine Learning	Algorithm	Parameter Control	Spark
K-means clustering	Machine Learning	Algorithm	Parameter Control	Spark
Logistic Regression	Machine Learning	Algorithm	Parameter Control	Spark
Alternating Least Squares(ALS)	Machine Learning	Algorithm	Parameter Control	Spark
Gradient Boosting Trees (GBT)	Machine Learning	Algorithm	Parameter Control	Spark
Linear Regression	Machine Learning	Algorithm	Parameter Control	Spark
Latent Dirichlet Allocation	Machine Learning	Algorithm	Parameter Control	Spark
Principal Components Analysis (PCA)	Machine Learning	Algorithm	Parameter Control	Spark
Random Forest	Machine Learning	Algorithm	Parameter Control	Spark
Support Vector Machine (SVM)	Machine Learning	Algorithm	Parameter Control	Spark
Support Vector Machine(SVM)	Machine Learning	Algorithm	Parameter Control	Spark
Singular Value Decomposition	Machine Learning	Algorithm	Parameter Control	Spark
Scan, Join, Aggregate	SQL	EO	Pre-Specified Process	Hadoop, Spark
PageRank	Websearch	Algorithm	Parameter Control	Spark
Nutch indexing	Websearch	Algorithm	Parameter Control	Spark, Nutch
NWeight	Graph	Algorithm	Parameter Control	Spark(with GraphX or Pregel)
Identity	Streaming	Algorithm, IO	Parameter Control	Spark Streaming, Flink, Storm and Gearpump
Repartition	Streaming	Algorithm, IO	Parameter Control	Spark Streaming, Flink, Storm and Gearpump
Stateful Wordcount	Streaming	Algorithm, IO	Parameter Control	Spark Streaming, Flink, Storm and Gearpump
Fixwindow	Streaming	Algorithm, IO	Parameter Control	Spark Streaming, Flink, Storm and Gearpump

kata dalam inputnya. Data masukan untuk beban kerja Sort dan WordCount dihasilkan menggunakan program RandomTextWriter yang nantinya akan dibuat melalui proses *Data Generation*.

### 2.8.2 *Data Generation* pada *Word Count* dan *Sort*

*Data generation* merupakan tahapan krusial dalam benchmark menggunakan HiBench, khususnya untuk beban kerja *Word Count* dan *Sort*. Tahapan ini bertanggung jawab untuk membentuk data acak yang akan diproses oleh kedua beban kerja tersebut. Tujuannya adalah untuk mensimulasikan skenario nyata dengan input data yang bervolume besar dan beragam.

Pada HiBench, skrip *prepare.sh* seperti pada Algoritma II.1 berperan penting dalam menyiapkan data untuk beban kerja. Skrip ini mengeksekusi program *randomtextwriter* yang terdapat dalam paket Hadoop. *randomtextwriter* menghasilkan sekumpulan data acak yang terdiri dari kata-kata yang diambil dari daftar kata yang telah ditentukan. Jumlah data yang dihasilkan, jumlah *map*, dan jumlah *reduce* dapat dikonfigurasi melalui parameter-parameter yang diberikan kepada skrip *prepare.sh*.

---

**Listing II.1** Skrip yang Digunakan HiBench pada Tahap *Data Generation*

---

```
1 #!/bin/bash
2
3 current_dir=`dirname "$0"`
4 current_dir=`cd "$current_dir"; pwd`
5 root_dir=${current_dir}/../../../../..
6 workload_config=${root_dir}/conf/workloads/micro/sort.conf
7 . "${root_dir}/bin/functions/load_bench_config.sh"
8
9 enter_bench HadoopPrepareSort ${workload_config} ${current_dir}
10 show_bannar start
11
12 rmr_hdfs ${INPUT_HDFS} || true
13 START_TIME=`timestamp`
14
15 run_hadoop_job ${HADOOP_EXAMPLES_JAR} randomtextwriter \
16     -D mapreduce.randomtextwriter.totalbytes=${DATASIZE} \
17     -D mapreduce.randomtextwriter.bytespermap=$(( ${DATASIZE} / ←
18         ${NUM_MAPS} )) \
19     -D mapreduce.job.maps=${NUM_MAPS} \
20     -D mapreduce.job.reduces=${NUM_RED} \
21     ${INPUT_HDFS}
22 END_TIME=`timestamp`
23 show_bannar finish
24 leave_bench
```

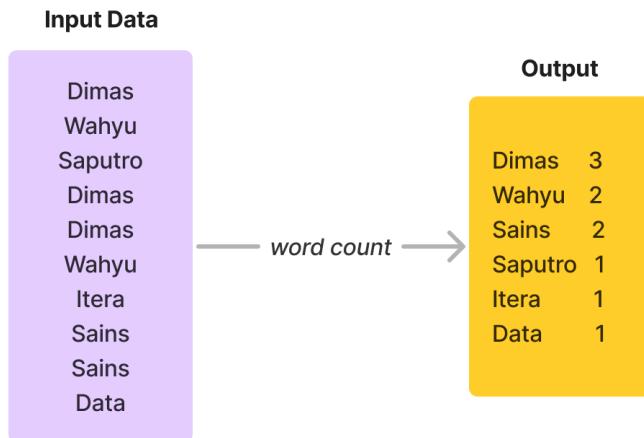
---

### 2.8.3 Beban Kerja *Word Count*

*Word Count* adalah algoritma sederhana untuk membaca berkas teks, dan menghitung jumlah kemunculan kata-kata pada file tersebut. Pada algoritma ini, inputnya berupa berkas teks dan outputnya berupa pasangan kata-kata dan jumlah kemunculannya. Beban kerja *word count* akan menghasilkan data keluaran yang lebih kecil dari pada data input. Karena itu, *word count* memiliki sifat *CPU Bound* yang nantinya akan ditandai dengan tingkat penggunaan CPU yang tinggi dan penggunaan I/O ringan. Selain itu, perlakunya diperkirakan akan tetap sama bahkan pada cluster yang lebih besar.

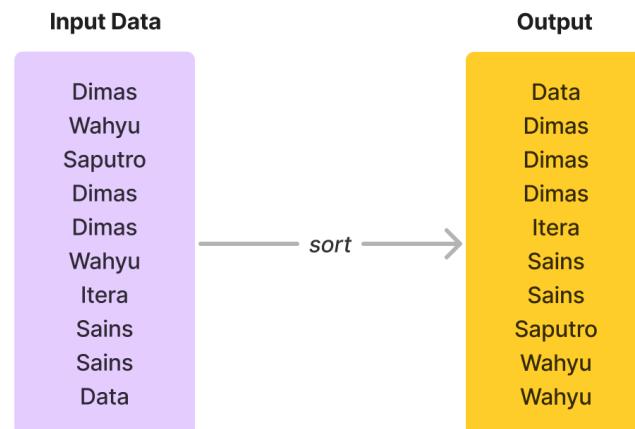
### 2.8.4 Beban Kerja *Sort*

*Sort* adalah algoritma yang umum digunakan untuk mengurutkan data berdasarkan kriteria tertentu. Algoritma ini menerima data dalam bentuk acak sebagai input, dan menghasilkan data yang terurut sebagai output. Data input dan output memi-



**Gambar 2.13** Contoh Input dan Output *Word Count*

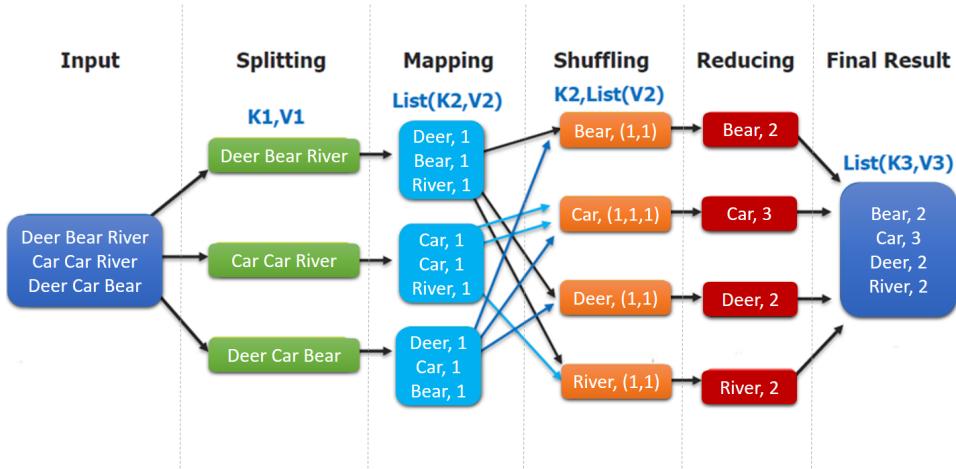
liki ukuran yang sama, sehingga beban kerja sort tidak menghasilkan pengurangan data. Kompleksitas algoritma *sort* bervariasi, tetapi umumnya membutuhkan perbandingan dan pertukaran elemen data yang intensif. Oleh karena itu, beban kerja *sort* cenderung bersifat *I/O bound*, dengan pemanfaatan CPU yang rendah dan penggunaan I/O yang tinggi.



**Gambar 2.14** Contoh Input dan Output *Sort*

Implementasi MapReduce pada *Word Count*[8] dapat dilihat pada Gambar 2.15. Pada proses MapReduce, data masukan akan melalui beberapa tahapan pemrosesan. Pertama, data akan dipecah menjadi bagian-bagian yang lebih kecil pada proses pemecahan data masukan (*splitting*). Dalam kasus Hadoop MapReduce, data idealnya akan dipecah menjadi beberapa blok berukuran maksimal 128MB.

Kemudian, blok data tersebut akan diproses lebih lanjut pada tahap pemetaan (*mapping*). Pemetaan merupakan salah satu tahapan terpenting dalam MapReduce. Pada



Gambar 2.15 Implementasi MapReduce pada Word Count [45]

tahap ini, blok data yang sudah dipecah akan diproses untuk menghasilkan pasangan kunci-nilai (*key-value pairs*) sementara, seperti pada contoh kasus *wordcount* yang menghasilkan pasangan kunci-nilai *Dear:1*, *Bear:1*, dan *River:1*. Pemetaan dapat melibatkan satu atau beberapa mesin pekerja (*worker*) yang memproses blok data secara paralel.

Selanjutnya adalah tahap pengocokan (*shuffling*) di mana pasangan kunci-nilai hasil pemetaan yang tersebar di beberapa mesin akan dikumpulkan berdasarkan kesamaan kuncinya agar bisa diproses lebih lanjut. Misalnya semua pasangan dengan kunci *Bear* dikumpulkan dalam satu mesin.

Pada tahap terakhir yaitu pengurangan (*reducing*), dilakukan agregasi terhadap pasangan kunci-nilai dengan kunci yang sama untuk menghasilkan keluaran akhir. Seperti pada contoh kasus *wordcount*, pasangan *Bear:1* dan *Bear:1* akan dijumlahkan menjadi *Bear:2* oleh proses pengurangan.

## 2.9 Data Keluaran HiBench dan Dool

Diagram pada Gambar 2.16 mengilustrasikan data keluaran dari dua alat, yaitu HiBench dan Dool, yang digunakan untuk mengevaluasi kinerja sistem. HiBench berfokus pada pengukuran kinerja keseluruhan dari suatu beban kerja (*benchmark*), sementara Dool memberikan pemantauan sistem yang terperinci secara *real-time*. HiBench menghasilkan laporan yang mencakup dua metrik utama, yaitu

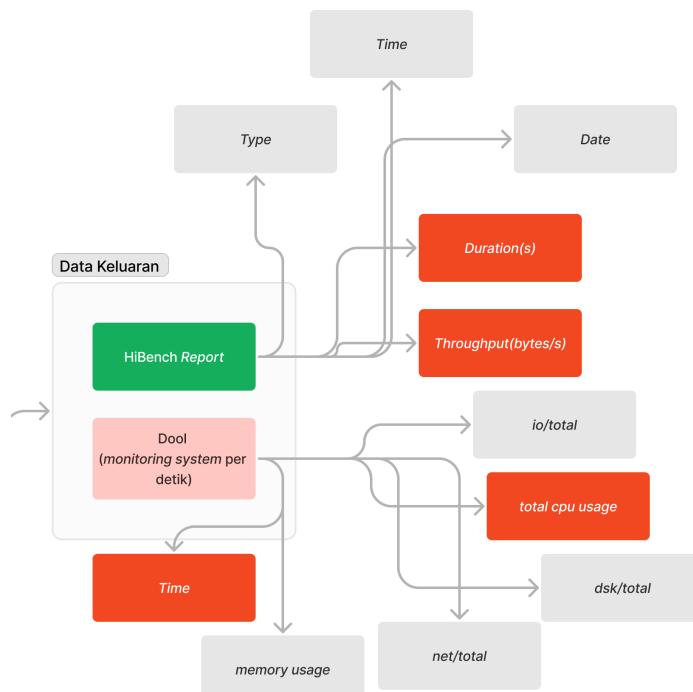
- Waktu Eksekusi (Execution Time):** Menunjukkan total waktu yang dibutuhkan untuk menjalankan beban kerja (*benchmark*) dari awal hingga akhir, diukur dalam detik.
- Throughput:** Mengukur jumlah data yang diproses per satuan waktu, biasanya dalam bita (*byte*) per detik. Metrik ini mencerminkan efisiensi sistem

dalam menangani beban kerja.

Dool menyediakan pemantauan sistem yang mendetail dan diperbarui setiap detik. Data keluaran Dool meliputi berbagai aspek kinerja sistem, antara lain:

1. **Time**: Timestamp yang menunjukkan waktu pengambilan data.
2. **Date**: Tanggal pengambilan
3. **Type**: Jenis pengukuran yang dilakukan, misalnya CPU, memori, disk, atau jaringan.
4. **io/total**: Total aktivitas input/output (I/O) pada *disk*, diukur dalam jumlah operasi I/O per
5. **total cpu usage**: Persentase penggunaan CPU secara keseluruhan.
6. **dsk/total**: Total aktivitas *disk*, mencakup baca dan tulis, diukur dalam bita per detik.
7. **memory usage**: Jumlah memori yang sedang digunakan oleh sistem.
8. **net/total**: Total aktivitas jaringan, mencakup data yang dikirim dan diterima, diukur dalam bita per detik.

Akhirnya, HiBench memberikan gambaran umum tentang efisiensi sistem dalam menangani beban kerja tertentu, sementara Dool memungkinkan kita untuk memantau berbagai komponen sistem secara *real-time* dan mengidentifikasi potensi *bottleneck* atau masalah kinerja.



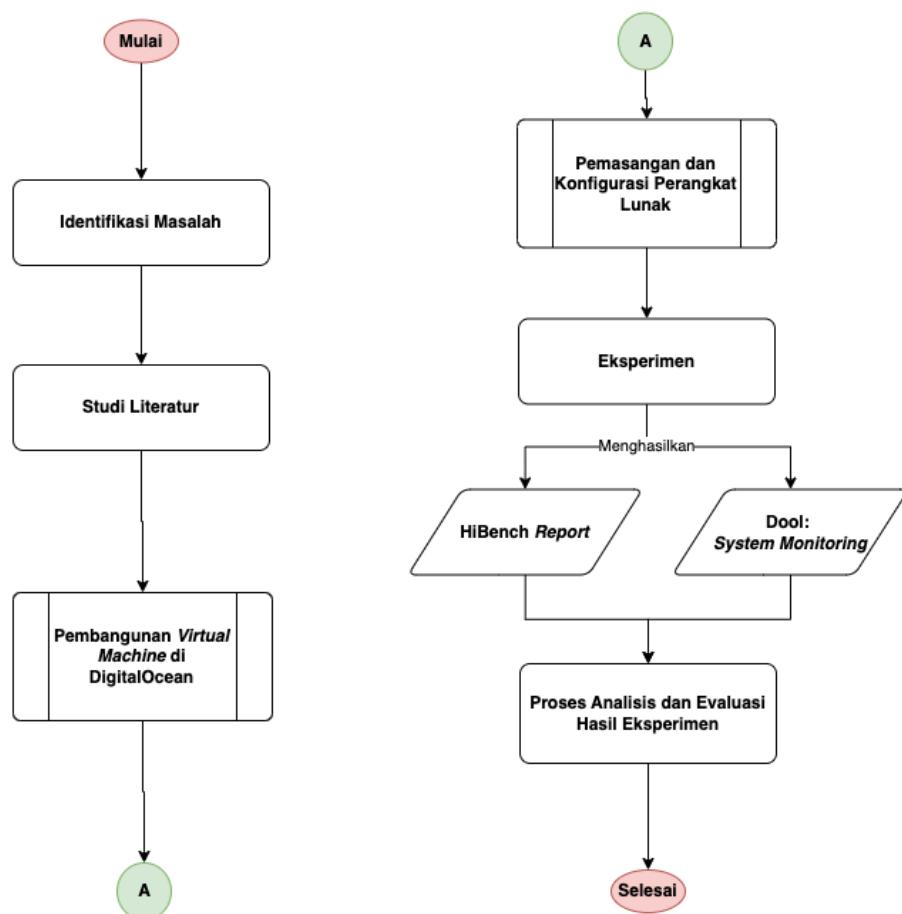
Gambar 2.16 Data Keluaran HiBench dan Dool

## BAB III

# METODOLOGI PENELITIAN

### 3.1 Alur Penelitian

Adapun diagram alir penelitian ini ditunjukkan pada Gambar 3.1 terdapat enam tahapan. Langkah awal yang dilakukan pada penelitian ini adalah melakukan identifikasi masalah, yaitu proses mencari, menghimpun, serta menemukan permasalahan yang nantinya akan diselesaikan. Setelah melakukan identifikasi masalah, langkah selanjutnya adalah studi literatur. Studi literatur adalah tahapan untuk mencari solusi dari permasalahan yang sebelumnya sudah kita definisikan. Pencarian solusi ini dapat melalui membaca referensi ilmiah terdahulu, baik melalui jurnal, buku, dokumentasi resmi, tesis, dan lain-lain. Tahapan ini akan memberikan pemahaman mendasar mengenai permasalahan yang sudah didapatkan sebelumnya.



Gambar 3.1 Diagram Alir Penelitian

Kemudian, penelitian ini akan dilanjutkan pada tahap membangun *virtual machine* di DigitalOcean. DigitalOcean adalah perusahaan penyedia layanan awan *Infrastructure as a Service* (IaaS) yang memberikan banyak pilihan kepada pengguna untuk menggunakan berbagai jenis layanan sesuai dengan kebutuhan, salah satunya yaitu *virtual machine*. *Virtual Machine* tersebut dapat dihentikan atau dihapus kapanpun saat tidak lagi diperlukan. Ketika infrastruktur sudah siap digunakan, penelitian dilanjutkan ke tahap pemasangan perangkat lunak, seperti Hadoop, Spark, dan Hi-Bench. Selanjutnya dilakukan eksperimen pada beban kerja *Micro Benchmarks*. Akhirnya, hasil dari eksperimen akan digunakan untuk proses analisis dan evaluasi.

### **3.2 Identifikasi Masalah dan Studi Literatur**

Langkah awal penelitian ini adalah identifikasi masalah dan studi literatur. Identifikasi masalah dapat dipahami sebagai tahapan mendefinisikan masalah sehingga masalah tersebut dapat terukur dan jelas untuk dijadikan landasan dalam latar belakang penelitian. Setelah masalah berhasil diidentifikasi, langkah selanjutnya adalah studi literatur yang mana dalam proses ini dilakukan pengumpulan berbagai macam informasi, referensi, dan konsep dasar yang menjadi landasan dasar dari penelitian. Langkah ini dapat dilakukan melalui membaca artikel ilmiah pendukung, buku-buku yang ditulis oleh para ahli, dan jika berkaitan dengan pemrograman dapat melihat dari dokumentasi resmi. Pada tahap ini juga dilakukan analisis terhadap penelitian terdahulu dan dibandingkan dengan identifikasi masalah yang didapatkan untuk membuka celah penelitian baru sehingga penelitian ini dapat bermanfaat.

### **3.3 Membangun *Virtual Machine* di DigitalOcean**

Konfigurasi perangkat keras merupakan aspek penting dalam mengevaluasi kinerja aplikasi *big data* berbasis Hadoop dan Spark. DigitalOcean, sebagai penyedia layanan infrastruktur sebagai layanan (IaaS), memberikan pengguna kebebasan penuh untuk membuat, mengkonfigurasi, dan mengelola berbagai infrastruktur yang telah disediakan. Dalam konteks penelitian ini, diperlukan penggunaan mesin virtual, yang dalam DigitalOcean dikenal sebagai "*Droplets*," yang memungkinkan untuk menyesuaikan berbagai aspek seperti sistem operasi, kapasitas penyimpanan, jumlah prosesor, dan parameter lainnya sesuai dengan kebutuhan spesifik penelitian. Penelitian ini mengadopsi mode *pseudo-distributed* yang memungkinkan penggunaan hanya satu *virtual machine* dalam konfigurasi *single node*. Walaupun hanya menggunakan satu *virtual machine*, mode *pseudo-distributed* memungkinkan setiap proses dalam klaster beroperasi secara independen, menciptakan lingkungan di mana semua proses berjalan mandiri satu sama lain. Hal ini memungkinkan untuk

**Tabel 3.1** Konfigurasi Perangkat Keras

Nama Parameter	Nilai Parameter
<b>Lokasi Pusat Data</b>	Singapore - Datacenter 1 - SGP1
<b>Sistem Operasi</b>	Ubuntu 20.04 (LTS) x64
<b>Jenis Droplet</b>	Basic
<b>Prosesor</b>	Premium AMD - 4 Core
<b>Memori</b>	8 GB
<b>Penyimpanan</b>	160 GB NVMe SSD

lebih berfokus pada pengumpulan data dan analisis, tanpa perlu melakukan konfigurasi yang rumit terkait dengan pengaturan klaster. Spesifikasi perangkat keras yang digunakan untuk *virtual machine* dalam mode *pseudo-distributed* sesuai pada Tabel 3.1. Penjelasan lengkap tentang pembuatan *virtual machine* (VM) pada *platform* DigitalOcean dan cara mengakses VM tersebut disajikan pada Lampiran A.

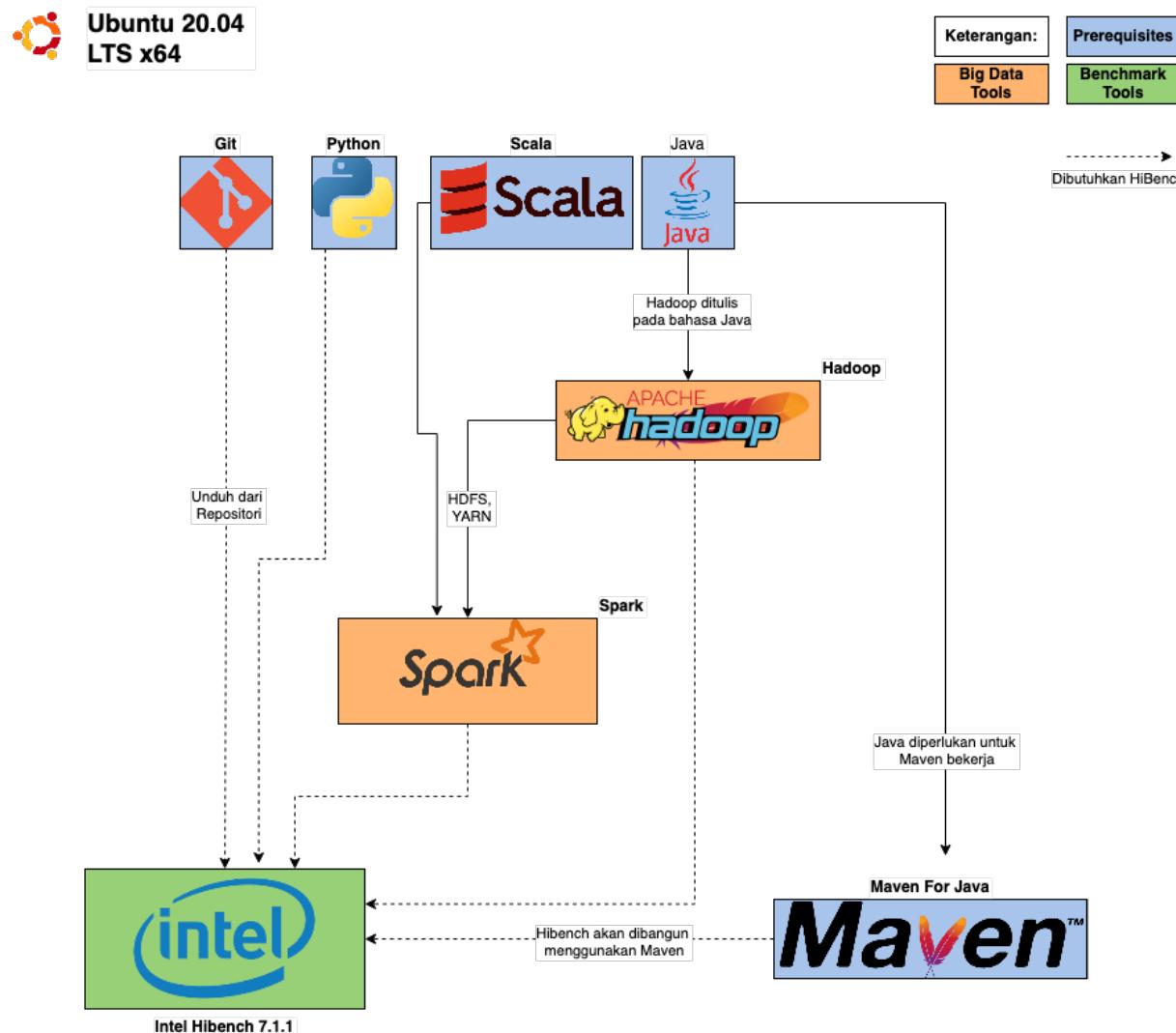
### 3.4 Pemasangan dan Konfigurasi Perangkat Lunak

Pemasangan dan konfigurasi perangkat lunak merupakan hal yang krusial dalam penelitian ini. Perangkat lunak yang diperlukan ditunjukkan pada Tabel 3.2.

**Tabel 3.2** Perangkat Lunak yang Dibutuhkan

Perangkat Lunak	Deskripsi
Ubuntu 20.04 LTS x64	Sistem operasi Linux berbasis Ubuntu
Git	Sistem kontrol versi untuk mengelola perubahan dalam kode sumber perangkat lunak
Maven	Perangkat lunak manajemen proyek Java
Java 8	
Python 3.7	Bahasa pemrograman dasar
Scala 2.11.8	
Hadoop 2.4	Perangkat lunak pengolahan data terdistribusi untuk penyimpanan dan manajemen data besar
Spark 2.1.3	Kerangka kerja pemrosesan data terdistribusi yang berjalan di atas Hadoop
Hibench	Alat yang digunakan untuk mengukur kinerja Hadoop dan Spark
Dool	Alat untuk melihat penggunaan <i>resource</i> sistem

Alur kerja instalasi perangkat lunak dalam penelitian ini dapat dilihat pada Gambar 3.2. Pada gambar, terdapat tiga bagian utama, yaitu *prerequisites* (perangkat lunak prasyarat) ditandai dengan warna biru, alat penyimpanan dan pemrosesan *Big Data* ditandai dengan warna oranye, dan alat untuk mengukur kinerja *big data* ditandai dengan warna hijau.



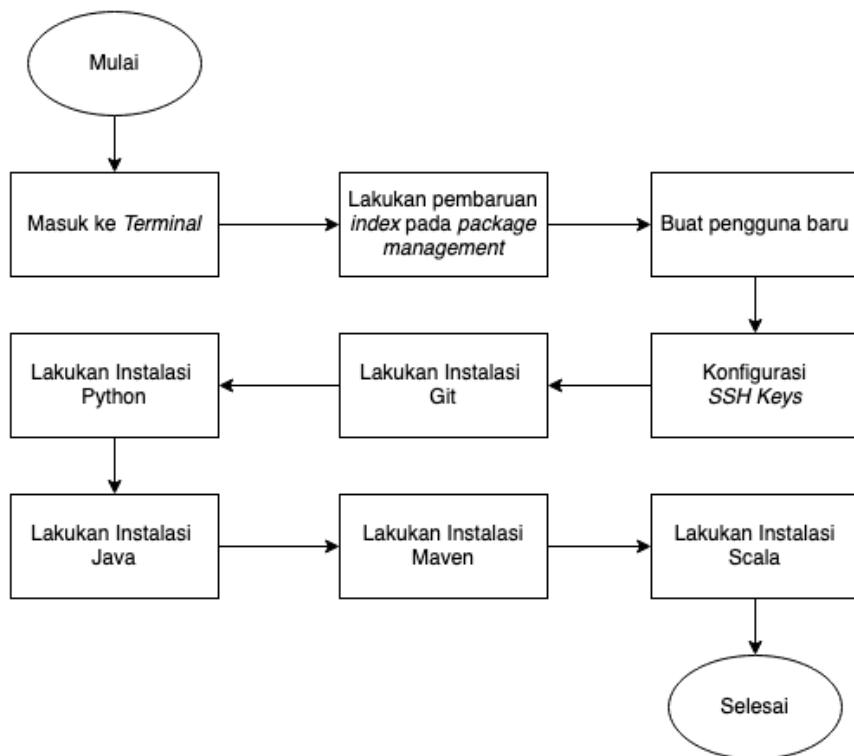
Gambar 3.2 Alur Instalasi Perangkat Lunak

### 3.4.1 Instalasi Perangkat Lunak Prasyarat

Ada beberapa perangkat lunak yang perlu diimplementasikan sebelum memasang Hadoop, Spark, dan HiBench, yaitu:

1. Ubuntu 20.04 LTS x64
2. Git
3. Java 8 dan Maven
4. Python 3.7
5. Scala 2.11.8

Pemasangan dan konfigurasi perangkat lunak pada tahapan ini tidak membutuhkan urutan. Akan tetapi, pada penelitian ini dibuatkan alur untuk pemasangan dan konfigurasi perangkat lunak prasyarat seperti pada Gambar 3.3. Penjelasan lengkap mengenai tata cara instalasi dan konfigurasi perangkat lunak prasyarat ini disajikan pada Lampiran B.

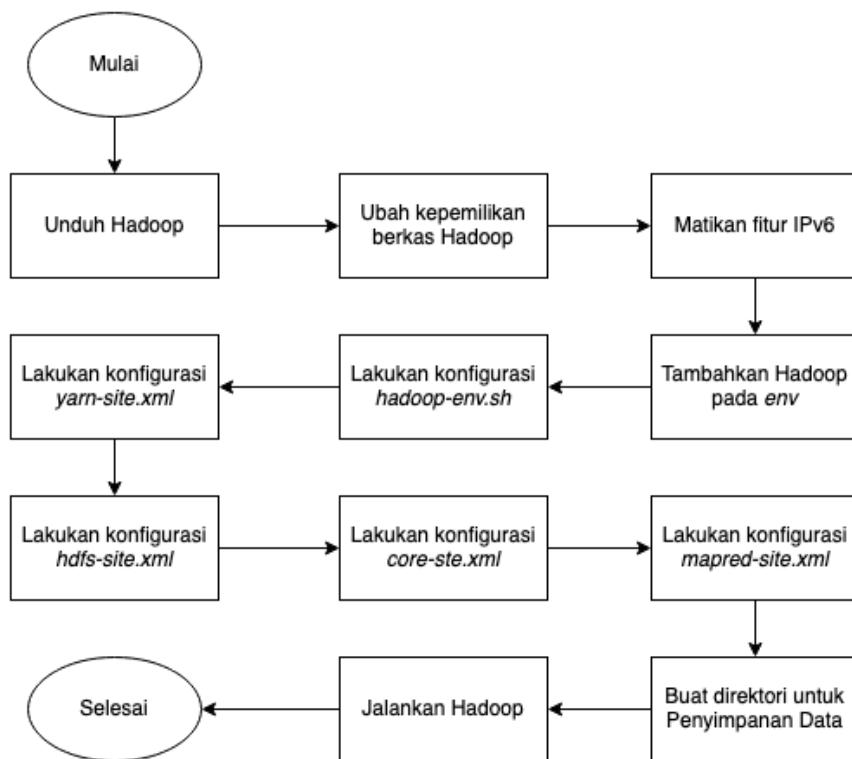


Gambar 3.3 Alur Instalasi Perangkat Lunak Prasyarat

### 3.4.2 Instalasi dan Konfigurasi Hadoop

Hadoop adalah perangkat lunak *open source* yang efektif dalam menyimpan dan memproses data dalam skala besar. Daripada menggunakan satu komputer besar

untuk menyimpan dan memproses data, Hadoop memungkinkan pengklasteran beberapa komputer untuk menganalisis set data besar secara paralel dengan lebih cepat. Ada beberapa perangkat lunak prasyarat yang perlu dipasang sebelum menggunakan Hadoop. Setelah perangkat lunak prasyarat berhasil dipasang, Hadoop juga dapat dipasang mengikuti panduan lengkap pada Lampiran C. Secara umum, alur yang harus dilakukan meliputi pengunduhan berkas Hadoop. Selanjutnya akan dilakukan pengubahan kepemilikan berkas ke *user hdfsuser*. Karena Hadoop tidak mendukung IPv6, maka fitur ini perlu dimatikan juga. Alur pemasangan dan konfigurasi Hadoop lebih jelas sesuai dengan Gambar 3.4.

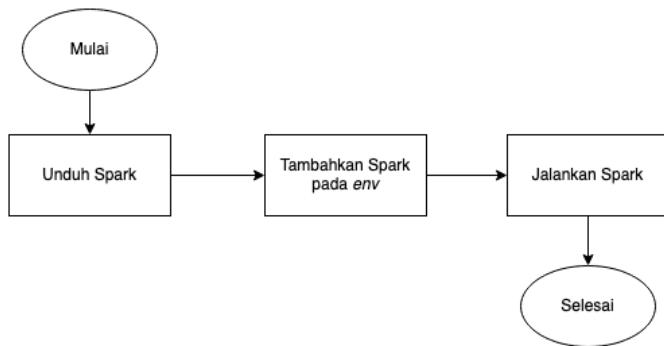


Gambar 3.4 Alur Instalasi dan Konfigurasi Hadoop

### 3.4.3 Instalasi dan Konfigurasi Spark

Apache Spark adalah sebuah kerangka kerja pengolahan data terdistribusi yang sangat cepat dan efisien. Spark dan Hadoop memiliki hubungan yang erat. Spark dapat berjalan di atas *Hadoop Distributed File System* (HDFS) dan dapat menggunakan Hadoop YARN sebagai manajer sumber daya. Oleh karena itu, instalasi Spark membutuhkan Hadoop sudah terpasang lebih dahulu. Alur pemasangan dan konfigurasi spark terlihat seperti pada Gambar 3.5. Apabila Hadoop sudah berhasil

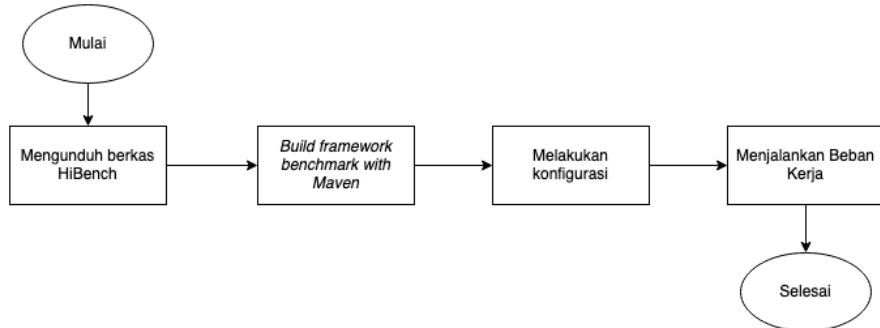
terpasang, langkah selanjutnya adalah memasang Spark seperti pada Lampiran D.



**Gambar 3.5** Alur Instalasi dan Konfigurasi Spark

#### 3.4.4 Instalasi dan Konfigurasi HiBench

Sebelum melakukan eksperimen, diperlukan suatu perangkat lunak pengukuran kinerja sistem *Big Data*, yaitu HiBench. HiBench tidak dapat digunakan secara langsung ketika sudah berhasil diunduh, melainkan harus dilakukan pembangunan beberapa modul yang dibutuhkan dengan Maven dan konfigurasi beberapa parameter.



**Gambar 3.6** Alur Instalasi dan Konfigurasi HiBench

Secara umum, alur instalasi dan konfigurasi HiBench sesuai dengan Gambar 3.6. Berkas HiBench diunduh dari repositori, dilanjutkan dengan pembangunan beberapa modul yang nantinya dibutuhkan. Selanjutnya, dilakukan konfigurasi beberapa berkas seperti *hibench.conf*, *hadoop.conf*, dan *spark.conf*. Jika telah dilakukan konfigurasi, dapat dilanjutkan dengan menjalankan beban kerja atau eksperimen. Lebih lanjut, pemasangan dan konfigurasi HiBench dijelaskan pada Lampiran E.

### 3.5 Eksperimen

Setelah instalasi dan konfigurasi perangkat keras dan perangkat lunak berhasil diselesaikan, tahap selanjutnya adalah eksperimen. Tahap ini melibatkan serangkaian pengujian yang terkontrol untuk mengevaluasi kinerja *platform big data* Hadoop dan Spark dalam menjalankan beban kerja tertentu dengan berbagai ukuran data. Tujuan utama eksperimen ini adalah untuk menjawab pertanyaan penelitian yang telah didefinisikan sebelumnya dan memperoleh pemahaman yang komprehensif tentang karakteristik kinerja masing-masing *platform*.

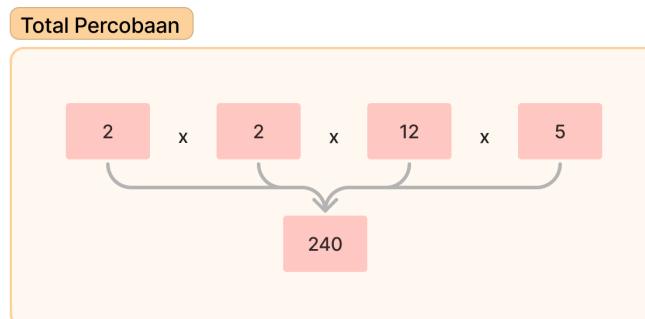
Penelitian ini difokuskan pada pengujian dua beban kerja yang umum dalam pemrosesan big data, yaitu *word count* dan *sort*. Beban kerja ini akan dieksekusi pada dua *platform big data* yang populer, yaitu Hadoop dan Spark. Setiap kombinasi *platform* dan beban kerja akan diuji dengan 12 ukuran data yang berbeda, mulai dari 100 KB hingga 15 GB. Detail ukuran data yang digunakan ditunjukkan pada Tabel 3.3. Untuk memastikan reliabilitas dan konsistensi hasil, setiap kombinasi *platform*, beban kerja, dan ukuran data akan diulang sebanyak 5 kali.

Proses eksperimen menghasilkan dua jenis berkas data:

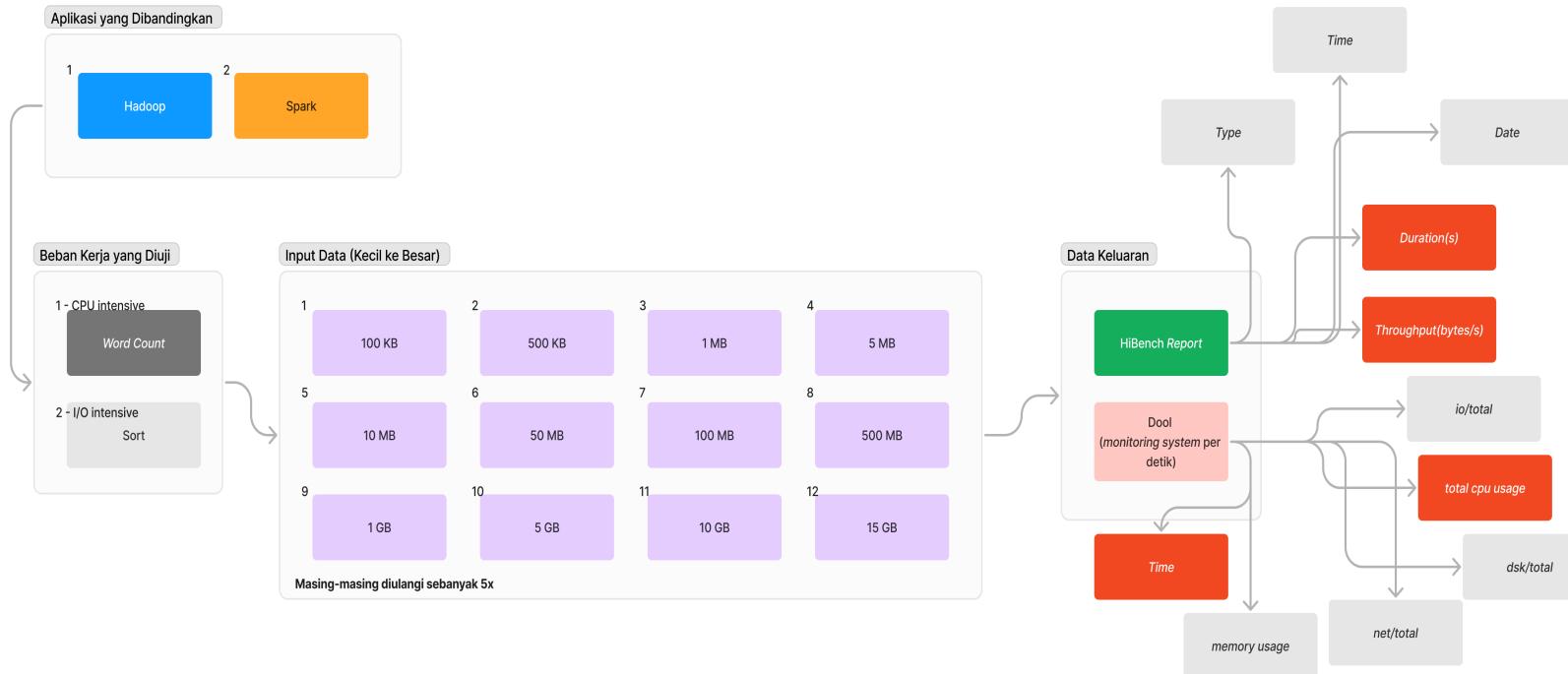
1. *HiBench Report*: Berisi informasi tentang kinerja beban kerja, termasuk waktu eksekusi, dan *throughput*.
2. *Dool System Monitoring*: Berisi informasi detail tentang aktivitas sistem selama eksekusi beban kerja, seperti penggunaan CPU, memori, I/O disk, dan jaringan.

Secara keseluruhan, desain eksperimen ini menghasilkan 240 percobaan individu, seperti yang diilustrasikan pada Gambar 3.7. Setiap percobaan mewakili kombinasi unik dari:

1. *Platform big data* (Hadoop atau Spark)
2. Beban kerja (*word count* atau *sort*)
3. Ukuran data (12 variasi)
4. Pengulangan (5 kali)



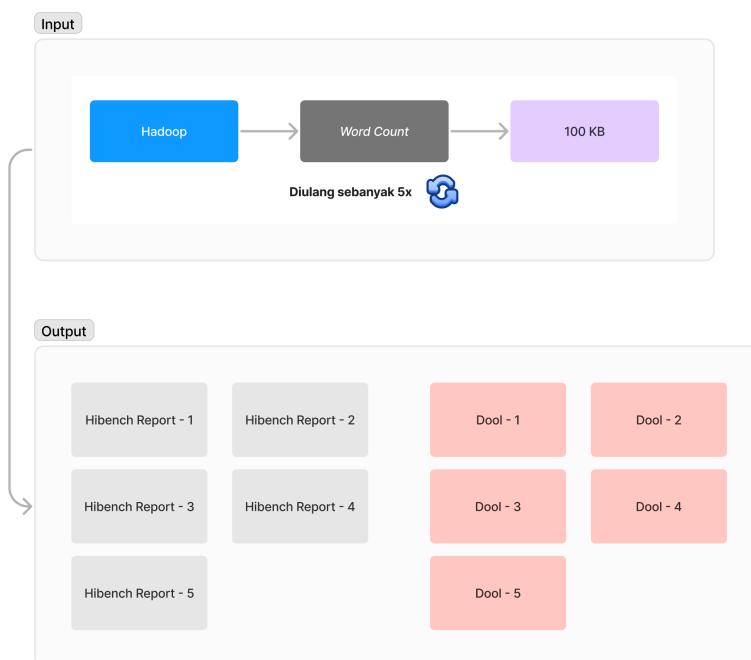
Gambar 3.7 Total Percobaan

**Gambar 3.8** End-to-end Penelitian

**Tabel 3.3 Variasi Input Data**

No	Label Input Data	Ukuran Input Data (bita)
1	100 KB	$100000 (1 * 10^5)$
2	500 KB	$500000 (5 * 10^5)$
3	1 MB	$1 * 10^6$
4	5 MB	$5 * 10^6$
5	10 MB	$1 * 10^7$
6	50 MB	$5 * 10^7$
7	100 MB	$1 * 10^8$
8	500 MB	$5 * 10^8$
9	1 GB	$1 * 10^9$
10	5 GB	$5 * 10^9$
11	10 GB	$1 * 10^{10}$
12	15 GB	$1.5 * 10^{10}$

Sebagai contoh, untuk *platform* Hadoop dengan beban kerja *word count* dan ukuran data 100 KB, akan menghasilkan lima HiBench *Report* dan lima berkas Dool *System Monitoring*, sesuai dengan jumlah pengulangan. Ilustrasi ini dapat dilihat pada Gambar 3.9.



**Gambar 3.9** Contoh Percobaan

Karena jumlah percobaan yang banyak, otomatisasi menjadi penting untuk memastikan efisiensi dan akurasi. Skrip khusus dikembangkan untuk mengotomatiskan seluruh proses eksperimen, termasuk konfigurasi HiBench, persiapan data, eksekusi beban kerja, dan pengumpulan data. Detail skrip otomatisasi dapat ditemukan

pada Lampiran F.

Algoritma otomatisasi eksperimen dimulai dengan mengubah direktori kerja ke direktori HiBench. Selanjutnya, algoritma melakukan iterasi untuk setiap beban kerja yang ditentukan. Di dalam setiap iterasi beban kerja, dilakukan iterasi lagi untuk setiap ukuran data. Pada setiap kombinasi beban kerja dan ukuran data, konfigurasi HiBench diubah sesuai dengan ukuran data yang dipilih.

Skrip persiapan data Hadoop dan Spark dijalankan berulang kali hingga proses persiapan berhasil. Setelah data siap, perulangan dilakukan sebanyak jumlah pengulangan yang ditentukan. Dalam setiap perulangan, perangkat lunak "dool" diaktifkan untuk memonitor aktivitas sistem, *benchmark* Hadoop atau Spark dijalankan, dan monitoring sistem dihentikan.

Setelah semua perulangan selesai, algoritma menunggu selama 15 detik sebelum melanjutkan ke ukuran data berikutnya. Proses ini berlanjut hingga semua kombinasi beban kerja dan ukuran data selesai diproses.

### 3.6 Analisis dan Evaluasi Hasil Eksperimen

Setelah menyelesaikan 240 percobaan yang dijelaskan di bagian eksperimen, langkah selanjutnya adalah menganalisis dan mengevaluasi hasil yang diperoleh. Analisis ini bertujuan untuk menjawab pertanyaan penelitian dan memahami bagaimana kinerja Hadoop dan Spark dalam menjalankan beban kerja *word count* dan *sort* dengan berbagai ukuran data. Berikut adalah beberapa aspek yang akan dikaji:

#### 1. Kinerja

- (a) **Persebaran Waktu Eksekusi pada Hadoop dan Spark.** Bagian ini akan menganalisis sebaran waktu eksekusi untuk setiap beban kerja (*word count* dan *sort*) pada kedua aplikasi (Hadoop dan Spark) dengan berbagai ukuran data. Analisis ini akan membantu memahami variabilitas kinerja dan konsistensi hasil pada setiap kombinasi aplikasi, beban kerja, dan ukuran data.
- (b) **Persebaran Throughput pada Hadoop dan Spark.** Mirip dengan analisis waktu eksekusi, persebaran *throughput* juga akan dianalisis untuk setiap kombinasi aplikasi, beban kerja, dan ukuran data. Throughput, yang menunjukkan jumlah data yang diproses per satuan waktu, merupakan metrik penting dalam evaluasi kinerja sistem big data. Visualisasi distribusi throughput akan membantu dalam memahami efisiensi pemrosesan data oleh Hadoop dan Spark.
- (c) **Rata-rata Waktu Eksekusi pada Hadoop dan Spark.** Selain persebaran, rata-rata waktu eksekusi untuk setiap kombinasi akan dihitung

dan dibandingkan. Ini memberikan gambaran umum tentang kinerja relatif setiap aplikasi dalam menyelesaikan beban kerja tertentu dengan ukuran data tertentu. Perbedaan rata-rata waktu eksekusi antara Hadoop dan Spark, serta tren perubahannya seiring dengan peningkatan ukuran data, akan diidentifikasi dan dibahas.

- (d) **Rata-rata Throughput pada Hadoop dan Spark.** Serupa dengan rata-rata waktu eksekusi, rata-rata *throughput* juga akan dihitung dan dibandingkan untuk setiap kombinasi. Analisis ini membantu memahami bagaimana efisiensi pemrosesan data berubah dengan berbagai beban kerja dan ukuran data, serta memberikan wawasan tentang skalabilitas setiap platform.
- (e) **Rate of Change.** *Rate of Change* akan dihitung untuk metrik-metrik kinerja seperti waktu eksekusi dan *throughput*. Analisis ini akan menunjukkan seberapa besar perubahan kinerja seiring dengan peningkatan ukuran data.

## 2. Penggunaan Sumber Daya

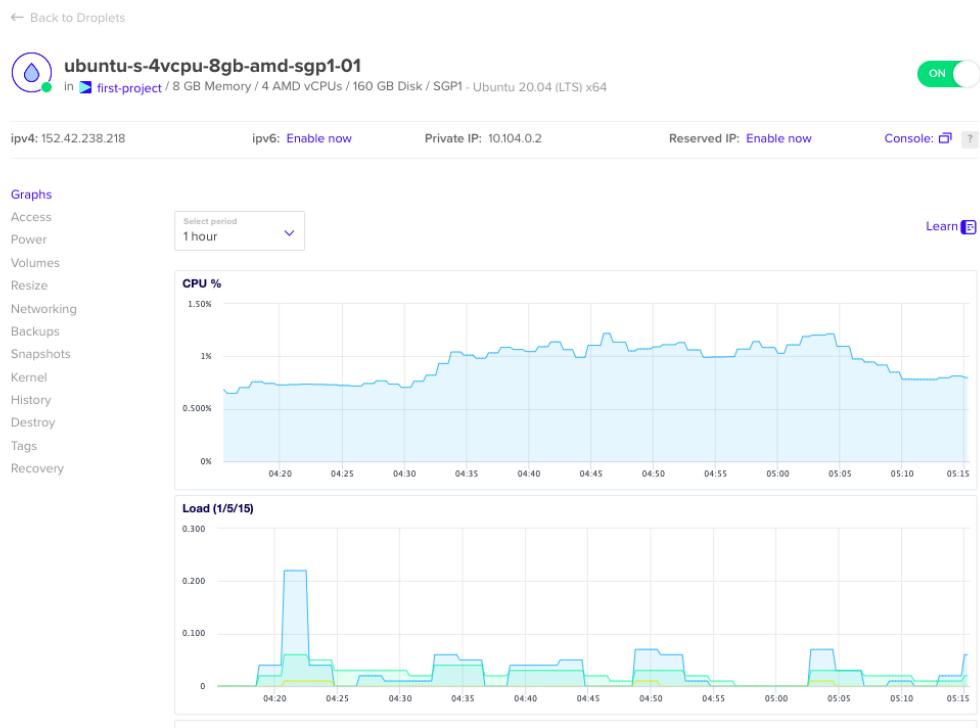
- (a) **Penggunaan CPU.** Bagian ini akan menganalisis penggunaan CPU oleh Hadoop dan Spark selama menjalankan berbagai beban kerja. Informasi ini dapat diperoleh dari berkas monitoring sistem yang dihasilkan oleh Dool. Analisis penggunaan CPU membantu memahami bagaimana setiap platform memanfaatkan sumber daya komputasi dan mengidentifikasi potensi optimasi.
- (b) **Utilisasi Sistem.** Selain penggunaan CPU, metrik-metrik lain seperti penggunaan memori, dan I/O penyimpanan. Hal ini memberikan gambaran yang lebih komprehensif tentang bagaimana setiap *platform* memanfaatkan sumber daya sistem dan potensi *bottleneck* yang mungkin terjadi selama pemrosesan data besar.

## BAB IV

# HASIL DAN PEMBAHASAN

### 4.1 Pembangunan *Virtual Machine* (VM) di DigitalOcean

Pembangunan *virtual machine* pada penelitian ini adalah hal yang krusial karena semua komputasi akan dijalankan pada *platform cloud* DigitalOcean. VM yang sudah berhasil terinisiasi akan terlihat seperti pada Gambar 4.1.



Gambar 4.1 Tampilan Dasbor VM DigitalOcean

### 4.2 Pemasangan dan Konfigurasi Perangkat Lunak

Penelitian ini membandingkan kinerja Hadoop dan Spark pada *platform cloud* DigitalOcean menggunakan alat pengujian data besar yang bernama HiBench pada lingkup *Micro Benchmarks*, yaitu *Word Count* dan *Sort* dengan data masukan berupa teks yang dibuat oleh *data generation* pada tahap persiapan.

Sebelum memulai eksperimen, serangkaian pemeriksaan dilakukan untuk memastikan bahwa semua perangkat lunak yang terlibat berfungsi dengan baik. Tahapan ini penting untuk menjamin validitas hasil penelitian. Berikut adalah pemeriksaan yang dilakukan, yaitu

1. **Pengecekan versi Hadoop.** Versi Hadoop yang digunakan dalam penelitian

ini adalah 2.4.0. Verifikasi versi dilakukan melalui perintah *hadoop version*, seperti yang ditunjukkan pada Gambar 4.2.

```
hadoop@ubuntu-s-4vcpu-8gb-amd-sgp1-01:~$ hadoop version
Hadoop 2.4.0
Subversion http://svn.apache.org/repos/asf/hadoop/common -r 1583262
Compiled by jenkins on 2014-03-31T08:29Z
Compiled with protoc 2.5.0
From source with checksum 375b2832a6641759c6eaf6e3e998147
This command was run using /srv/hadoop-2.4.0/share/hadoop/common/hadoop-common-2.4.0.jar
```

## Gambar 4.2 Pengecekan Versi Hadoop

2. **Pengecekan versi Spark.** Versi Spark yang digunakan adalah 2.1.3. Verifikasi dilakukan dengan perintah *spark-submit –version*, seperti yang ditunjukkan pada Gambar 4.3.

**Gambar 4.3** Pengecekan Versi Spark

3. **Pemeriksaan service yang berjalan ketika beban kerja belum dijalankan.** Status layanan (*services*) yang berjalan pada komputer diperiksa dalam keadaan tanpa beban kerja (*idle*). Layanan yang diharapkan aktif meliputi: Jps, ResourceManager, DataNode, NodeManager, NameNode, dan SecondaryNameNode. Gambar 4.4 menunjukkan hasil pemeriksaan layanan dasar.

```
hadoop@ubuntu-s-4vcpu-8gb-amd-sgp1-01:~$ jps  
1161637 Jps  
111512 ResourceManager  
111147 DataNode  
111851 NodeManager  
110973 NameNode  
111358 SecondaryNameNode
```

**Gambar 4.4** Pengecekan *Service* yang Berjalan (Normal)

4. **Pemeriksaan service yang berjalan ketika menggunakan Hadoop.** Ketika beban kerja Hadoop dijalankan, layanan tambahan seperti YarnChild, MRAppMaster, dan RunJar diharapkan aktif, di samping layanan dasar yang telah disebutkan. Gambar 4.5 menunjukkan hasil pemeriksaan layanan saat Hadoop aktif.

```
hadoop@ubuntu-s-4vcpu-8gb-amd-sgp1-01:~$ jps
1162768 YarnChild
1162898 YarnChild
1162837 YarnChild
1162425 MRAppMaster
111147 DataNode
111851 NodeManager
1162623 YarnChild
1162143 RunJar
1162690 YarnChild
1162567 YarnChild
111512 ResourceManager
110973 NameNode
111358 SecondaryNameNode
1163022 Jps
```

Gambar 4.5 Pengecekan Service yang Berjalan (Hadoop)

5. **Pemeriksaan service yang berjalan ketika menggunakan Spark.** Ketika beban kerja Spark dijalankan, layanan seperti CoarseGrainedExecutorBackend, ExecutorLauncher, dan SparkSubmit diharapkan aktif, di samping layanan dasar. Gambar 4.6 menunjukkan hasil pemeriksaan layanan saat Spark aktif.

```
hadoop@ubuntu-s-4vcpu-8gb-amd-sgp1-01:~$ jps
1164818 CoarseGrainedExecutorBackend
1165141 Jps
1164742 ExecutorLauncher
1164009 SparkSubmit
111512 ResourceManager
111147 DataNode
111851 NodeManager
110973 NameNode
111358 SecondaryNameNode
```

Gambar 4.6 Pengecekan Service yang Berjalan (Spark)

### 4.3 Eksperimen

Selama pengujian, beberapa parameter pada HiBench, Hadoop, dan Spark dikonfigurasi secara tetap untuk menjaga konsistensi dan memungkinkan perbandingan yang adil. Tabel 4.1 dan 4.2 merangkum konfigurasi parameter yang digunakan.

**Tabel 4.1** Konfigurasi HiBench

Nama Parameter	Nilai	Keterangan Parameter
hibench.default.map.parallelism	8	Mapper numbers (Hadoop), partition numbers (Spark)
hibench.default.shuffle.parallelism	8	Reducer numbers (Hadoop), shuffle partition (Spark)

**Tabel 4.2** Konfigurasi Spark

Nama Parameter	Nilai Parameter	Keterangan Parameter
hibench.yarn.executor.num	2	Jumlah <i>executor</i>
hibench.yarn.executor.cores	4	Jumlah <i>core CPU</i> setiap <i>executor</i>
spark.executor.memory	4G	Jumlah memori setiap <i>executor</i>
spark.driver.memory	4G	Jumlah memori tiap <i>driver</i> Spark

Parameter *hibench.default.map.parallelism* memiliki peran yang berbeda pada Hadoop dan Spark. Pada Hadoop, parameter ini menentukan jumlah *Mapper*, yaitu proses yang bertanggung jawab untuk memproses data secara paralel pada tahap *Map*. Pada Spark, parameter ini menentukan jumlah partisi data, yaitu unit pemrosesan dasar dalam Spark.

Parameter *hibench.default.shuffle.parallelism* juga memiliki peran yang berbeda pada Hadoop dan Spark. Pada Hadoop, parameter ini menentukan jumlah *Reducer*, yaitu proses yang bertanggung jawab untuk menggabungkan hasil dari tahap *Map*. Pada Spark, parameter ini menentukan jumlah *Shuffle partition*, yaitu jumlah partisi data yang digunakan selama tahap *Shuffle*, yaitu proses pengocokan dan pengurutan data antara tahap *Map* dan *Reduce*.

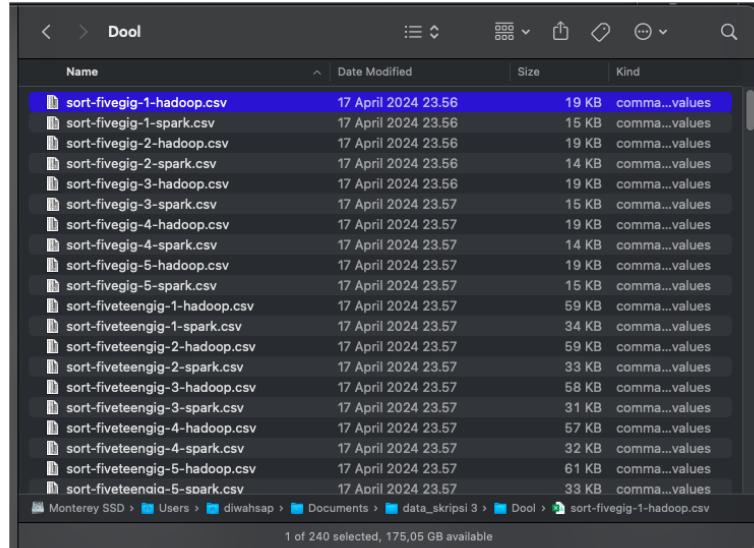
### 4.4 Data Keluaran yang Dihasilkan

Setiap pengujian akan menghasilkan berkas output berupa data HiBench *Report* dan *Dool System Monitoring*. Data HiBench *Report* akan terlihat seperti pada Gambar 4.7. Pada Gambar tersebut, terlihat bahwa ekstensi berkasnya *.report* dan terlihat beberapa data seperti jenis beban kerja, aplikasi yang digunakan, besar input data, durasi, dan *throughput*.

Dool, alat monitoring sistem, menghasilkan berkas CSV (*comma-separated value*) untuk setiap perulangan eksperimen. Dengan demikian, terdapat sekitar 240 berkas CSV, seperti yang ditunjukkan pada Gambar 4.8. Penamaan berkas mengikuti format: [jenis beban kerja]-[ukuran data]-[nomor perulangan]-[aplikasi]. Sebagai

Type	Date	Time	Input_data_size	Duration(s)	Throughput(bytes/s)	Throughput/node
HadoopWordCount	2024-04-17	08:14:56	106922	26.333	4060	4060
HadoopWordCount	2024-04-17	08:15:30	106922	31.216	3425	3425
HadoopWordCount	2024-04-17	08:15:59	106922	27.237	3925	3925
HadoopWordCount	2024-04-17	08:16:28	106922	26.381	4052	4052
HadoopWordCount	2024-04-17	08:16:58	106922	27.263	3921	3921
ScalaSparkWordCount	2024-04-17	08:17:57	105373	35.491	2969	2969
ScalaSparkWordCount	2024-04-17	08:18:36	105373	35.866	2937	2937
ScalaSparkWordCount	2024-04-17	08:19:14	105373	35.972	2929	2929
ScalaSparkWordCount	2024-04-17	08:19:52	105373	35.618	2958	2958
ScalaSparkWordCount	2024-04-17	08:20:30	105373	35.843	2939	2939
HadoopWordCount	2024-04-17	08:22:15	516076	28.427	18154	18154
HadoopWordCount	2024-04-17	08:22:45	516076	27.191	18979	18979
HadoopWordCount	2024-04-17	08:23:14	516076	27.210	18966	18966
HadoopWordCount	2024-04-17	08:23:43	516076	26.326	19603	19603
HadoopWordCount	2024-04-17	08:24:14	516076	28.275	18252	18252
ScalaSparkWordCount	2024-04-17	08:25:13	516295	35.679	14470	14470
ScalaSparkWordCount	2024-04-17	08:25:52	516295	35.831	14499	14499
ScalaSparkWordCount	2024-04-17	08:26:30	516295	35.895	14383	14383
ScalaSparkWordCount	2024-04-17	08:27:08	516295	35.860	14397	14397
ScalaSparkWordCount	2024-04-17	08:27:47	516295	35.686	14467	14467
HadoopWordCount	2024-04-17	09:43:36	107227	27.652	3877	3877
ScalaSparkWordCount	2024-04-17	09:44:35	107705	35.646	3021	3021
HadoopWordCount	2024-04-17	09:45:02	107705	25.173	4278	4278
ScalaSparkWordCount	2024-04-17	09:46:02	106302	35.550	2990	2990
HadoopWordCount	2024-04-17	09:46:30	106302	26.203	4055	4055
ScalaSparkWordCount	2024-04-17	09:47:01	106623	35.767	2981	2981
HadoopWordCount	2024-04-17	09:47:58	106623	25.153	4238	4238
HadoopWordCount	2024-04-17	10:07:03	106383	27.192	3912	3912

Gambar 4.7 Data HiBench Report



Gambar 4.8 Berkas Dool

contoh, *sort-fivegig-1-hadoop.csv* menunjukkan *data monitoring* untuk beban kerja sort, data masukan 5 GB, perulangan pertama, dan aplikasi Hadoop.

Struktur data Dool, yang ditunjukkan pada Gambar 4.9, mencakup baris *header* (baris 1-4) dan nama kolom (baris 6). Data ini akan dianalisis lebih lanjut untuk mendapatkan *insight* tentang kinerja Hadoop dan Spark.

sort-fivegig-1-hadoop.csv

Open with Microsoft Excel

dool 1.3.1 CSV output						URL:		https://github.com/scottchiefbaker/dool/																							
Author:	Scott Baker					User:		hadoop																							
Host:	ubuntu-s-4vcpu-8gb-amd-sgp1-01					Date:		17 Apr 2024 15:03:05 UTC																							
C cmdline: /home/hadoop/bin/dool --all --io --output sort-fivegig-1-hadoop.csv --bytes																															
total cpu usage						disk/total		net/total		paging		memory usage		system		procs		load avg		i		i		i		i		i			
usr	sys	idl	wai	stl	read	writ		recv	send	in	out	used	free	cach	aval	int	csw	run	bik	new	1m	5m	15m	r	1	0	8.155	3.650	3.560	3.590	4
23.761	2.013	74.113	0.049	0.064	5228437.613	16319861.692	0	0	0	0	2162262016	475381760	5444534272	5884563456	984.883	3074.792	1	0	46	3.650	3.560	3.590	6								
45.297	3.218	51.485	0	0	1048576	40960	330	1356	0	0	2161164288	475455480	544558272	588504544	3089	4605	1	0	46	3.650	3.560	3.590	6								
43.750	4	52	0	0.250	1048576	32768	264	1158	0	0	2165202944	470269952	544660648	5881778176	2892	4249	1	0	30	3.650	3.560	3.590	6								
60.759	10.127	24.051	5.063	0	6979584	286494720	198	966	0	0	2301308928	325865472	5454323712	5745946624	3928	6333	3	1	117	3.650	3.560	3.590	2								
29.250	1.500	67.250	2	0	1073152	91414528	528	1970	0	0	2338525184	287350784	5455609856	5708730368	2624	3814	0	0	20	3.360	3.500	3.570	1								
1	1	98	0	0	1064960	0	0	0	0	0	2081677312	543322112	5456494592	596560160	747	1222	0	0	2	3.360	3.500	3.570	1								
0.504	0.756	98.741	0	0	1171456	0	54	66	0	0	2081255424	542203904	5457690624	5966151680	697	1257	0	0	1	3.360	3.500	3.570	6								
33.002	5.459	61.538	0	0	61440	0	146	66	0	0	2148663296	473927680	5458157568	5898842112	3724	6342	3	0	168	3.360	3.500	3.570	5								
55.335	2.481	41.687	0.248	0.248	1662976	0	0	0	0	0	2236891136	384565248	5459808256	5810622464	2962	3831	4	0	33	3.360	3.500	3.570	1								
31.566	1.768	66.414	0.253	0	155648	1093632	0	0	0	0	2285436928	335728640	5460094976	5762093056	1699	2702	0	0	52	3.090	3.440	3.550	2								
2	0.250	97.750	0	0	0	0	66	516	0	0	2292400128	328761344	5460099702	5755129856	672	1201	0	0	0	3.090	3.440	3.550	0								
54.156	8.312	37.531	0	0	0	0	0	0	0	0	2476548096	142532608	5460652032	5570985984	2886	11054	13	0	307	3.090	3.440	3.550	0								
93.734	6.266	0	0	0	376832	0	0	0	0	0	2678603776	130801664	527694880	5368492032	1683	13052	16	0	3	3.090	3.440	3.550	5								
93.750	5.750	0.250	0	0.250	1110016	110247936	0	0	0	0	2809208832	137953280	5142167552	523703104	2196	14933	14	0	27	3.090	3.440	3.550	1								
96.750	3.250	0	0	0	1048576	679936	146	66	0	0	2911137792	137613312	5043146752	5134532608	1703	7673	9	0	34	3.800	3.590	3.600	8								
91.750	8	0.250	0	0	50798592	0	0	0	0	0	3351273472	124530688	4628041728	4690513920	2501	7755	14	0	54	3.800	3.590	3.600	4								
86.466	13.534	0	0	0	331198464	0	0	0	0	0	3741163520	125767680	4225154176	4292571136	5321	10527	16	0	10	3.800	3.590	3.600	2								
87.500	12.500	0	0	0	30109248	0	0	0	0	0	3738468320	123019264	427143008	4272178592	4532	7544	13	0	0	3.800	3.590	3.600	2								
72.843	23.604	2.030	1.523	0	103849984	437792768	0	0	0	0	373954752	120360960	4276752384	4291170304	2814	3733	3	5	0	3.800	3.590	3.600	8								
78.238	20.207	0.518	0.777	0.259	28708864	240549888	0	0	0	0	3513155884	321720320	4301340672	4517687296	2245	4029	9	0	11	4.860	3.810	3.670	2								
68.367	27.806	2.296	1.531	0	184320	514318336	66	290	0	0	3027361792	955604992	4153470976	5003681792	2646	6187	1	0	88	4.860	3.810	3.670	2								
67.168	4.511	28.321	0	0	1634304	0	66	290	0	0	2437586944	1543766016	4155039744	5593509888	3873	7491	6	0	70	4.860	3.810	3.670	1								
87.940	7.789	4.271	0	0	14692352	0	0	0	0	0	2710876160	1254203392	4170199040	5320057376	2570	11019	10	0	227	4.860	3.810	3.670	1								
94.750	5.250	0	0	0	96337920	0	0	0	0	0	2883100672	985124864	4267167744	5148053504	2242	11472	22	0	1	4.860	3.810	3.670	7								
90.727	9.273	0	0	0	40775680	466944	0	0	0	0	3094884352	650993664	4389711872	4936105984	2113	12918	11	0	30	5.430	3.950	3.720	3								
90.727	9.023	0	0	0.251	94515200	397021184	0	0	0	0	3177091072	401768448	4555644928	4853927936	2614	8096	15	0	23	5.430	3.950	3.720	7								
86	14	0	0	0	36384768	0	0	0	0	0	3560140800	253534202	4325302272	4471037952	2332	5864	9	0	39	5.430	3.950	3.720	2								
87	12.750	0.250	0	0	344440832	32768	0	0	0	0	3503181824	132292608	4509560832	4527476736	5355	10168	11	0	14	5.430	3.950	3.720	2								
82	17.750	0.250	0	0	109338624	133484544	0	0	0	0	3266285568	336080896	4544860160	4764880896	2951	5181	7	0	50	5.430	3.950	3.720	8								
81.612	14.610	2.771	1.008	0	80007168	297545728	66	290	0	0	3138931456	134598656	469139856	4711895040	3129	5778	8	0	9	6.040	4.100	3.770	6								
72.362	27.387	0.251	0	0	1776640	491831296	66	290	0	0	2633867264	1292259328	4227891200	5397549056	2731	6895	11	0	81	6.040	4.100	3.770	2								
87.657	9.320	3.023	0	0	43237376	0	0	0	0	0	2663243776	1220345856	4270653440	5368209408	3170	9321	8	0	183	6.040	4.100	3.770	3								

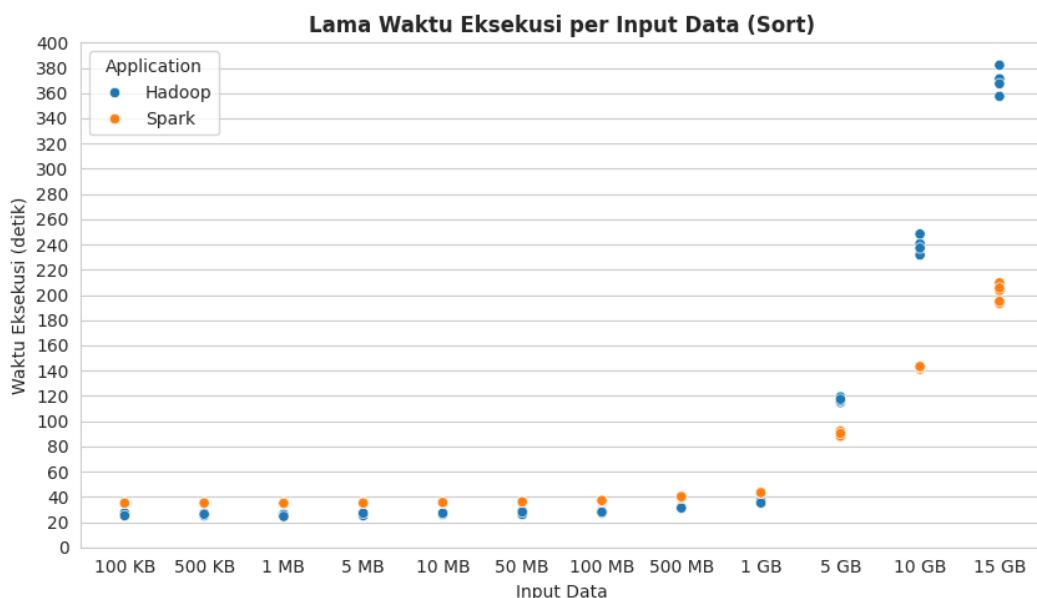
Gambar 4.9 Contoh Data Dool

## 4.5 Analisis dan Evaluasi Hasil Eksperimen: Kinerja

### 4.5.1 Persebaran Waktu Eksekusi pada Hadoop dan Spark

Waktu eksekusi adalah durasi yang diperlukan untuk memproses data. Nilai parameter ini diperoleh dengan menghitung selisih antara waktu awal dan waktu akhir saat Apache Hadoop dan Apache Spark dijalankan atau dihentikan untuk memproses input data dengan beban kerja masing-masing. Satuan pengukuran untuk parameter waktu eksekusi adalah detik. Setiap beban kerja dilakukan sebanyak lima kali pengulangan untuk mendapatkan hasil yang lebih akurat dan representatif.

Gambar 4.10 dan 4.11 menyajikan *scatter plot* yang membandingkan performa Hadoop dan Spark dalam dua tugas pemrosesan data yang berbeda, yaitu *sort* dan *word count*. Sumbu x pada kedua gambar menunjukkan variasi ukuran input data, mulai dari 100 KB hingga 15 GB, sementara sumbu y menunjukkan waktu eksekusi dalam detik.

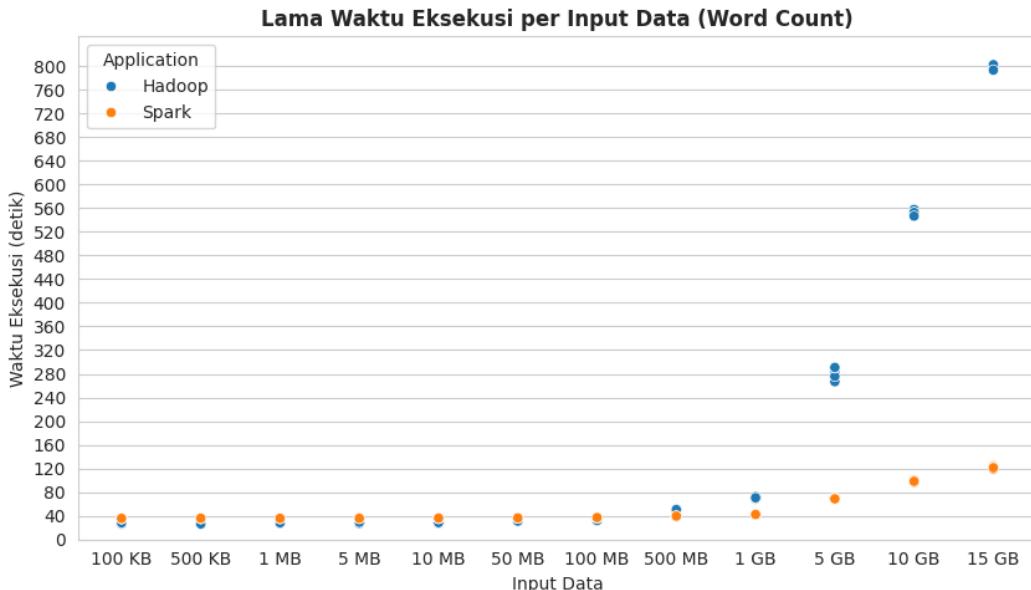


**Gambar 4.10** Persebaran Waktu Eksekusi *Sort* (Hadoop, Spark)

Pada Gambar 4.10, terlihat bahwa waktu eksekusi Hadoop untuk input data 100 KB hingga 1 GB secara konsisten lebih cepat dibandingkan Spark. Hadoop berada pada rentang waktu 20-40 detik, sedangkan Spark berada pada rentang waktu 35-45 detik.

Namun, untuk input data sebesar 5 GB, Spark menunjukkan waktu eksekusi yang lebih cepat dibandingkan Hadoop. Spark berada pada rentang 80-100 detik, sembari Hadoop berada pada rentang 110-125 detik. Perbedaan performa ini semakin signifikan seiring bertambahnya ukuran data, terutama pada ukuran data 10 GB dan 15 GB. Perbedaan waktu eksekusi antara Hadoop dan Spark semakin jauh pada

beban kerja *sort* dengan ukuran data yang lebih besar.



Gambar 4.11 Persebaran Waktu Eksekusi *Word Count* (Hadoop, Spark)

Pada Gambar 4.11, Spark menunjukkan performa yang lebih baik dibandingkan Hadoop untuk ukuran input data 500 MB, 1 GB, 5 GB, 10 GB, dan 15 GB pada beban kerja *word count*. Namun, untuk ukuran input data 100 KB hingga 100 MB, Hadoop masih lebih unggul. Waktu eksekusi Hadoop untuk input data 100 KB hingga 100 MB berada pada rentang 20-40 detik, meskipun perbedaannya tidak signifikan dibandingkan Spark karena titik data Hadoop dan Spark saling berdekatan.

Hasil ini menunjukkan bahwa Spark lebih unggul dan konsisten dibandingkan Hadoop dalam menangani tugas pemrosesan data yang lebih besar, dengan rincian sebagai berikut:

1. Untuk beban kerja *sort*, Spark lebih unggul mulai dari ukuran input data 5 GB, 10 GB, dan 15 GB.
2. Untuk beban kerja *word count*, Spark lebih unggul mulai dari ukuran input data 500 MB, 1 GB, 5 GB, 10 GB, dan 15 GB.

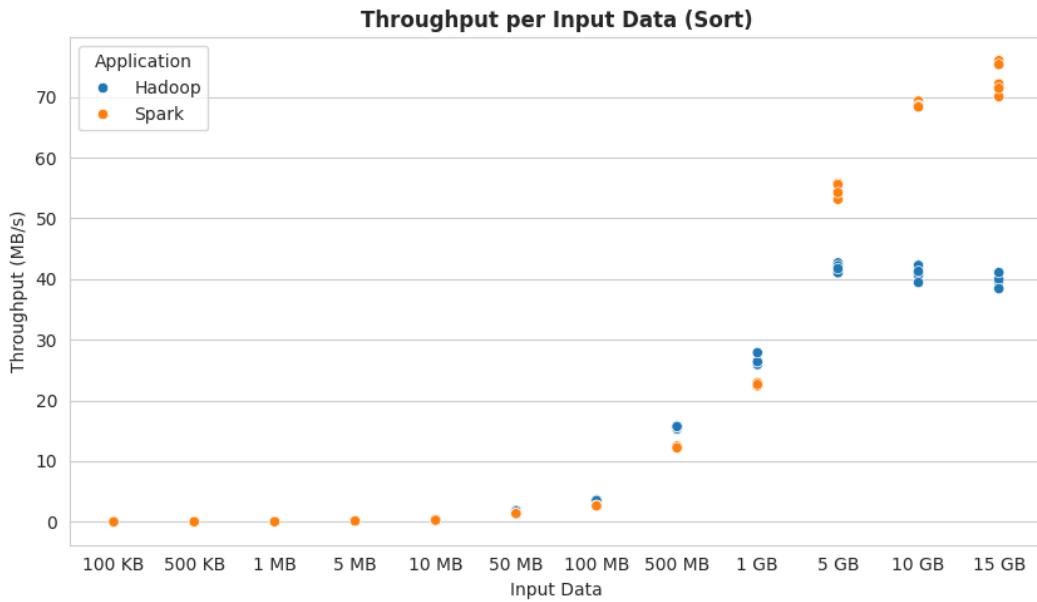
Secara keseluruhan, Spark menunjukkan kinerja yang lebih baik dalam menangani data berukuran besar, sementara Hadoop lebih efisien untuk data berukuran kecil hingga menengah.

#### 4.5.2 Persebaran *Throughput* pada Hadoop dan Spark

*Throughput* adalah kecepatan pertukaran data per detik. Kegiatan pertukaran data tersebut terjadi pada *node* yang dipakai dalam komputer komputasi, saat Hadoop maupun Spark memproses data. Oleh karena itu, semakin tinggi nilai *throughput*,

semakin sedikit waktu yang dibutuhkan untuk menyelesaikan komputasi. Satuan *throughput* pada penelitian ini adalah MB/s (mega bita per detik).

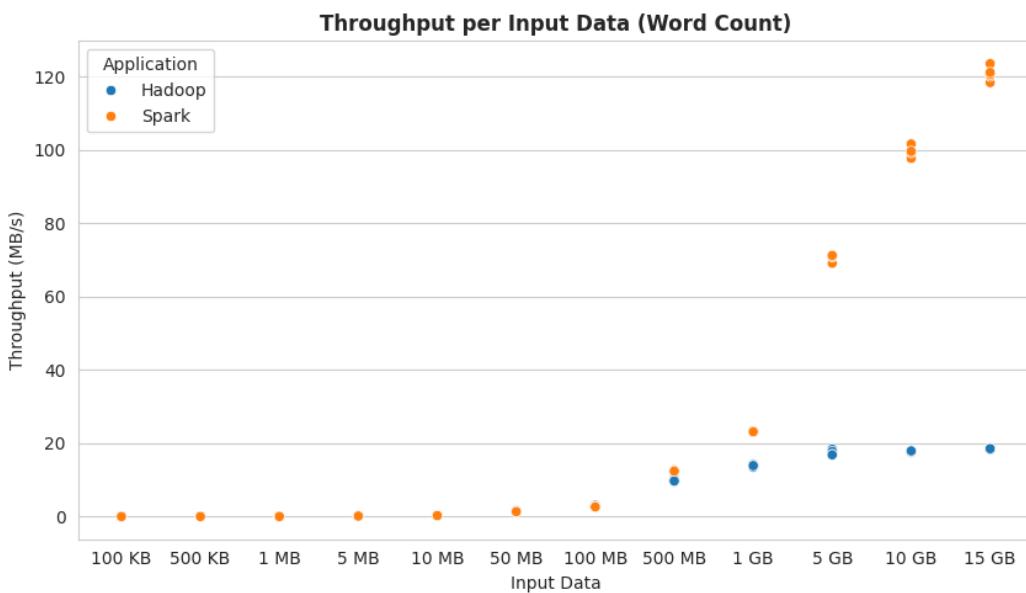
Gambar di bawah akan menyajikan *scatter plot* yang membandingkan *throughput* Hadoop dan Spark dalam dua tugas pemrosesan data, yaitu *sort* (Gambar 4.12) dan *word count* (Gambar 4.13). Sumbu x pada kedua gambar menunjukkan variasi ukuran input data, sedangkan sumbu y menunjukkan *throughput* dalam MB/s.



**Gambar 4.12** *Throughput Sort* (Hadoop, Spark)

Pada tugas *sort* (Gambar 4.12), Spark menunjukkan peningkatan *throughput* yang signifikan seiring dengan bertambahnya ukuran data. Pada ukuran data terbesar (15 GB), Spark mencapai *throughput* sekitar 70 MB/s. Sebaliknya, Hadoop menunjukkan peningkatan *throughput* yang lebih lambat dan hanya mencapai sekitar 40 MB/s pada ukuran data yang sama. Hal ini menunjukkan bahwa Spark mampu melakukan pertukaran data yang lebih besar daripada Hadoop.

Pada tugas *word count* (Gambar 4.13), Spark mencapai *throughput* yang lebih tinggi daripada Hadoop. Perbedaan *throughput* paling mencolok terlihat pada ukuran data terbesar (15 GB), di mana Spark mencapai *throughput* lebih dari 120 MB/s, sedangkan Hadoop hanya mencapai sekitar 20 MB/s. Meskipun Spark menunjukkan peningkatan *throughput* yang signifikan pada ukuran data besar (1 GB, 5 GB, 10 GB, dan 15 GB), pada data input yang lebih kecil, 100 KB sampai 100 MB, perbedaan *throughput* antara Hadoop dan Spark tidak berbeda jauh untuk *word count*. Hasil ini menunjukkan bahwa Spark lebih unggul dalam menangani data berukuran besar untuk kedua tugas pemrosesan data tersebut. Spark mampu mempertahankan *throughput* yang lebih tinggi dibandingkan Hadoop, terutama saat menangani data



Gambar 4.13 Throughput Word Count (Hadoop, Spark)

berukuran besar, dengan rincian sebagai berikut:

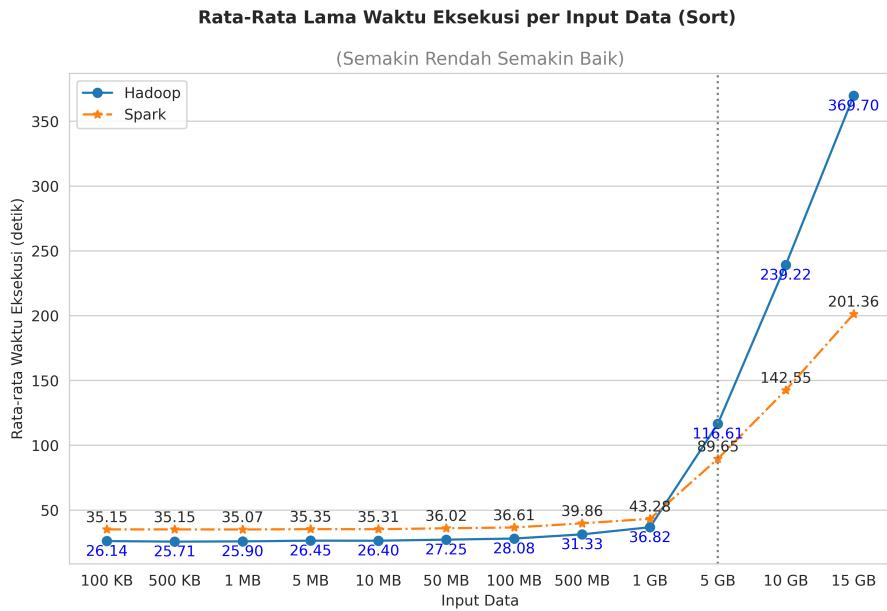
1. Untuk beban kerja *sort*, Spark lebih unggul pada ukuran data 1 GB, 5 GB, 10 GB, dan 15 GB.
2. Untuk beban kerja *word count*, Spark lebih unggul pada ukuran data 1 GB, 5 GB, 10 GB, dan 15 GB.

Secara keseluruhan, Spark menunjukkan kinerja yang lebih baik dalam hal *throughput* terutama pada data berukuran besar, sementara Hadoop masih menunjukkan performa yang kompetitif pada data berukuran kecil hingga menengah.

#### 4.5.3 Rata-rata Waktu Eksekusi pada Hadoop dan Spark

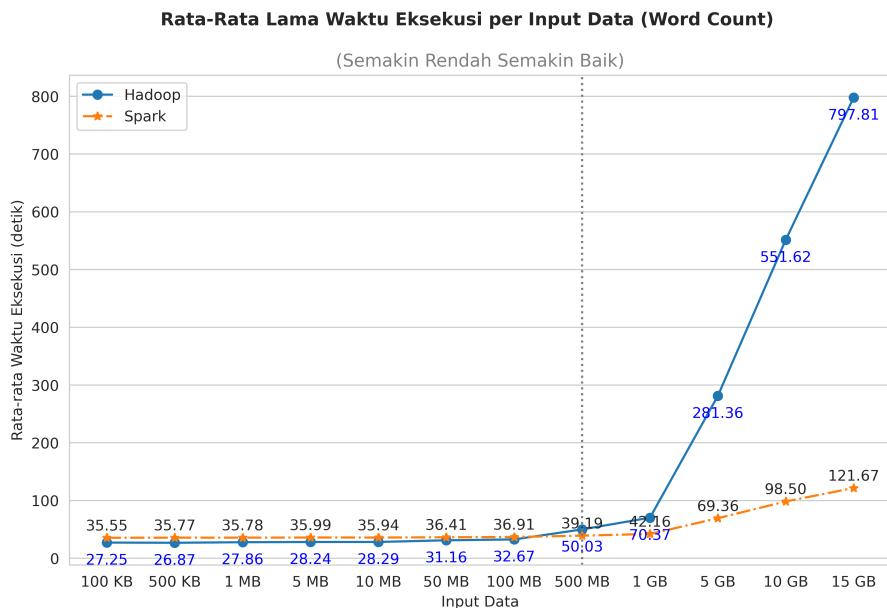
Gambar 4.14 dan 4.15 menyajikan *line plot* yang menggambarkan rata-rata waktu eksekusi Hadoop dan Spark untuk tugas *sort* dan *word count* dengan berbagai ukuran data. Sumbu x pada kedua gambar menunjukkan ukuran input data, sedangkan sumbu y menunjukkan rata-rata waktu eksekusi dalam detik. Garis vertikal pada kedua gambar menunjukkan titik di mana Spark mulai menunjukkan performa yang lebih cepat dibandingkan Hadoop.

Pada Gambar 4.14, terlihat bahwa Hadoop secara konsisten memiliki waktu eksekusi yang lebih rendah daripada Spark untuk ukuran input data 100 KB-1 GB pada tugas *sort*. Perbedaan performa mulai terlihat pada input data 5 GB, di mana waktu eksekusi pada Hadoop mulai meningkat secara eksponensial, sementara Spark menunjukkan kenaikan yang lebih moderat. Pada ukuran data terbesar (15 GB), waktu eksekusi Hadoop mencapai sekitar 369,70 detik, sedangkan Spark hanya mencapai



**Gambar 4.14** Rata-rata Waktu Eksekusi (*Sort*)

sekitar 201,36 detik.



**Gambar 4.15** Rata-rata Waktu Eksekusi (*Word Count*)

Pada Gambar 4.15, Spark juga menunjukkan performa yang lebih baik daripada Hadoop pada sebagian besar ukuran data pada tugas *word count*, khususnya mulai dari input data 500 MB sampai 15 GB. Lama waktu eksekusi beban kerja pada Hadoop mulai mengalami kenaikan sejak input data 500 MB. Sedangkan, Spark baru mengalami kenaikan waktu eksekusi pada input data 5 GB. Pada ukuran data terbesar (15 GB), waktu eksekusi Hadoop mencapai sekitar 797,81 detik, sedangkan

Spark hanya mencapai sekitar 121,67 detik.

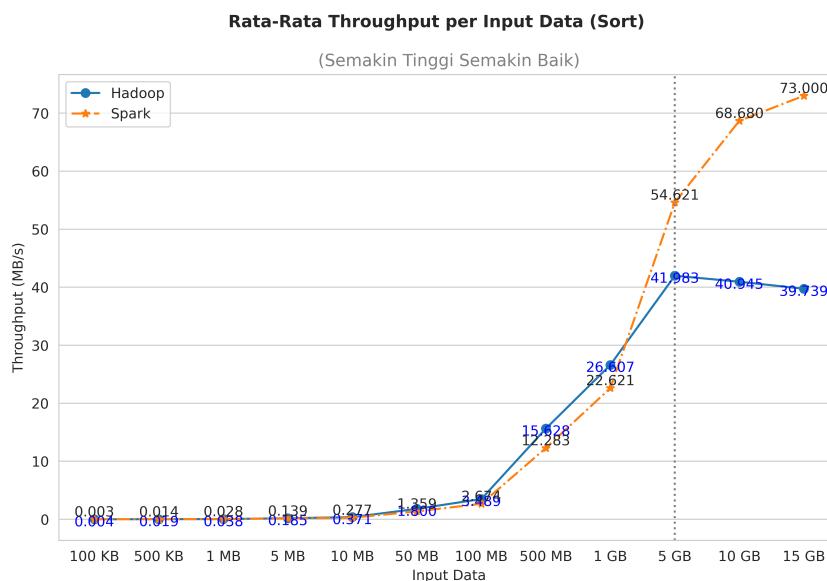
Hasil ini menunjukkan bahwa Spark lebih unggul dalam menangani data berukuran besar untuk kedua tugas pemrosesan data tersebut. Spark mampu mempertahankan waktu eksekusi yang lebih rendah dibandingkan Hadoop, terutama saat menangani data berukuran besar, dengan rincian sebagai berikut:

1. Untuk beban kerja *sort*, Hadoop lebih unggul pada ukuran data 100 KB hingga 1 GB, sedangkan Spark mulai unggul pada ukuran data 5 GB dan seterusnya.
2. Untuk beban kerja *word count*, Spark lebih unggul pada ukuran data 500 MB hingga 15 GB.

Secara keseluruhan, Spark menunjukkan kinerja yang lebih baik dalam hal waktu eksekusi terutama pada data berukuran besar, sementara Hadoop masih menunjukkan performa yang kompetitif pada data berukuran kecil hingga menengah.

#### 4.5.4 Rata-rata Throughput pada Hadoop dan Spark

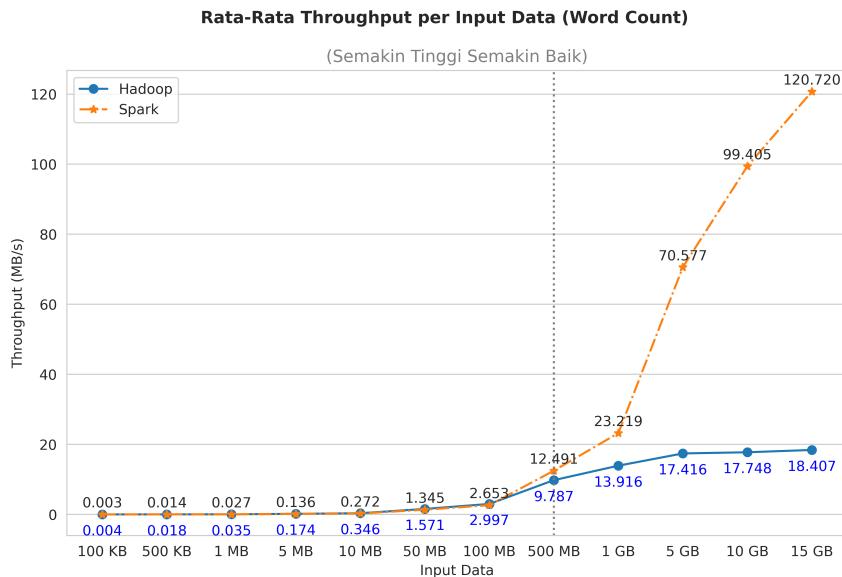
Gambar 4.16 dan 4.17 menyajikan *line plot* yang menggambarkan rata-rata throughput Hadoop dan Spark untuk tugas *sort* dan *word count* dengan berbagai ukuran data. Sumbu x pada kedua gambar menunjukkan ukuran input data, sedangkan sumbu y menunjukkan rata-rata *throughput* dalam MB/s. Garis vertikal pada kedua gambar menunjukkan titik di mana Spark mulai menunjukkan throughput yang lebih tinggi dibandingkan Hadoop.



**Gambar 4.16 Rata-rata Throughput (Sort)**

Pada beban kerja *sort*, Spark menunjukkan peningkatan *throughput* yang signifikan seiring dengan meningkatnya ukuran data. Pada ukuran data kecil (di bawah

10 MB), *throughput* Hadoop dan Spark relatif rendah dan sebanding. Setelah input data 10 MB, *throughput* pada Hadoop meningkat dan sedikit menjauhi Spark. Namun, setelah input data 1 GB, Spark secara konsisten menunjukkan *throughput* yang lebih tinggi, mencapai 73.00 MB/s pada 15 GB dibandingkan dengan 39.74 MB/s untuk Hadoop.



Gambar 4.17 Rata-rata *Throughput* (*Word Count*)

Pada beban kerja *word count*, Spark juga mengungguli Hadoop dalam hal *throughput* dengan perbedaan yang lebih besar daripada beban kerja *sort*. Awalnya, *throughput* Spark dan Hadoop sama dan saling berhimpit sejak input data 100 KB-100 MB. Spark mulai menunjukkan *throughput* yang lebih tinggi pada ukuran data 500 MB. Pada ukuran data 15 GB, Spark mencapai *throughput* 120.72 MB/s, sedangkan Hadoop hanya mencapai 18.41 MB/s.

Rata-rata *throughput* ini menunjukkan bahwa Spark lebih unggul dalam menangani data berukuran besar untuk kedua tugas pemrosesan data tersebut. Spark mampu mempertahankan *throughput* yang lebih tinggi dibandingkan Hadoop, terutama saat menangani data berukuran besar, dengan rincian sebagai berikut:

1. Untuk beban kerja *sort*, Hadoop lebih unggul pada ukuran data di bawah 1 GB, sedangkan Spark mulai unggul pada ukuran data 1 GB dan seterusnya.
2. Untuk beban kerja *word count*, Spark lebih unggul pada ukuran data 500 MB hingga 15 GB.

Secara keseluruhan, Spark menunjukkan kinerja yang lebih baik dalam hal *throughput* terutama pada data berukuran besar, sementara Hadoop masih menunjukkan performa yang kompetitif pada data berukuran kecil hingga menengah.

#### 4.5.5 Laju Perubahan (*Rate of Change*)

Laju perubahan (ROC) menggambarkan kecepatan perubahan suatu variabel dalam periode waktu tertentu. ROC diwakili oleh kemiringan garis dan sering diilustrasikan dengan huruf delta ( $\Delta$ ). Dalam konteks evaluasi performa sistem seperti Hadoop dan Spark, ROC dapat diaplikasikan untuk menganalisis:

1. **Execution Time.** ROC dari *execution time* menunjukkan seberapa cepat waktu eksekusi tugas berubah seiring dengan peningkatan beban kerja (misalnya, jumlah data yang diproses). Nilai ROC positif mengindikasikan peningkatan performa, sementara nilai negatif mengindikasikan penurunan performa. Artinya, sistem membutuhkan waktu lebih lama untuk memproses data seiring bertambahnya beban, mengindikasikan potensi bottleneck atau masalah skalabilitas.
2. **Throughput:** ROC dari *throughput* menunjukkan seberapa cepat laju pemrosesan data berubah seiring dengan peningkatan beban kerja. Nilai ROC positif mengindikasikan peningkatan performa, sementara nilai negatif mengindikasikan penurunan performa.

#### Laju Perubahan Waktu Eksekusi terhadap Input Data (*Sort*)

Gambar 4.18 menunjukkan laju perubahan waktu eksekusi Hadoop-Spark terhadap input data pada beban kerja *sort*. Hasil dan analisis dari gambar ini dapat dijelaskan sebagai berikut:

##### 1. 100 KB - 10 MB:

- *Spark:* Laju perubahan (ROC) pada data input 500 KB sampai 5 MB cenderung naik, dimulai dari nilai ROC pada 500 KB sebesar  $-5.4 \times 10^{-9}$  hingga mencapai  $6.7 \times 10^{-8}$  pada 5 MB. Namun, pada input data 10 MB, laju mengalami penurunan menjadi  $-8.4 \times 10^{-9}$ .
- *Hadoop:* ROC menunjukkan fluktuasi yang cukup signifikan. Pada 500 KB, nilai ROC adalah  $-1.1 \times 10^{-6}$ , kemudian naik drastis pada 5 MB menjadi  $3.9 \times 10^{-6}$  sebelum turun kembali pada 10 MB menjadi  $-9.7 \times 10^{-9}$ .

##### 2. 10 MB - 1 GB:

- *Spark:* Laju perubahan mengalami peningkatan dari 10 MB ke 50 MB ( $1.7 \times 10^{-8}$ ), kemudian sedikit menurun secara bertahap hingga 1 GB ( $6.7 \times 10^{-9}$ ).
- *Hadoop:* Pada rentang ini, nilai ROC mengalami penurunan secara bertahap dari 50 MB ke 1 GB, dimulai dari nilai ROC sebesar  $2.1 \times 10^{-6}$  pada 50 MB hingga mencapai  $1.1 \times 10^{-8}$  pada 1 GB.

### 3. 1 GB - 15 GB:

- *Spark*: Pada rentang ini, nilai ROC mulai dari 1 GB hingga 5 GB menunjukkan peningkatan yang signifikan, dari  $6.7 \times 10^{-9}$  menjadi  $1.1 \times 10^{-8}$ . Selanjutnya, pada input data tertinggi, yaitu 15 GB, nilai ROC mengalami sedikit penurunan.
- *Hadoop*: Nilai ROC pada input data 1 GB hingga 5 GB mengalami peningkatan yang drastis, mulai dari  $1.1 \times 10^{-8}$  menjadi  $1.9 \times 10^{-8}$ . Peningkatan ini berlanjut hingga 15 GB dengan nilai ROC mencapai  $2.5 \times 10^{-8}$ .

Analisis menunjukkan bahwa baik Spark maupun Hadoop menunjukkan tren laju perubahan yang bervariasi tergantung pada ukuran input data. Spark cenderung memiliki fluktuasi yang lebih halus, sedangkan Hadoop menunjukkan perubahan yang lebih drastis terutama pada ukuran input data yang lebih kecil.

### Laju Perubahan Waktu Eksekusi terhadap Input Data (*Word Count*)

Gambar 4.19 menunjukkan laju perubahan waktu eksekusi Hadoop-Spark terhadap input data pada beban kerja *word count*. Hasil dan analisis dari gambar ini dapat dijelaskan sebagai berikut:

#### 1. 100 KB - 10 MB:

- *Spark*: Laju perubahan (ROC) pada data input 500 KB mencapai puncaknya sebesar  $5.4 \times 10^{-7}$ , tetapi menurun drastis pada 1 MB menjadi  $2.1 \times 10^{-8}$ . Pada input data 5 MB dan 10 MB, laju perubahan terus menurun hingga mencapai  $-9.3 \times 10^{-9}$ .
- *Hadoop*: ROC juga menunjukkan fluktuasi yang signifikan dengan puncak pada 1 MB sebesar  $1.9 \times 10^{-6}$ . Setelah itu, ROC menurun pada 5 MB menjadi  $-9.3 \times 10^{-8}$  dan lebih lanjut pada 10 MB mencapai  $1.1 \times 10^{-8}$ .

#### 2. 10 MB - 1 GB:

- *Spark*: Laju perubahan mengalami peningkatan dari 10 MB ke 50 MB ( $1.1 \times 10^{-8}$ ) sebelum sedikit menurun pada 100 MB ( $-9.7 \times 10^{-9}$ ). Selanjutnya, ROC menunjukkan fluktuasi kecil pada 500 MB ( $5.6 \times 10^{-9}$ ) hingga 1 GB ( $5.8 \times 10^{-9}$ ).
- *Hadoop*: Pada rentang ini, nilai ROC menunjukkan fluktuasi dengan puncak pada 50 MB ( $7.0 \times 10^{-8}$ ) dan penurunan pada 100 MB ( $2.9 \times 10^{-8}$ ). Setelah itu, ROC stabil dari 500 MB ( $4.2 \times 10^{-8}$ ) hingga 1 GB ( $4.0 \times 10^{-8}$ ).

#### 3. 1 GB - 15 GB:

- *Spark*: Pada rentang ini, nilai ROC mulai dari 1 GB hingga 5 GB me-

nunjukkan peningkatan yang signifikan, dari  $5.8 \times 10^{-9}$  menjadi  $6.6 \times 10^{-9}$ . Namun, pada input data tertinggi, yaitu 15 GB, nilai ROC mengalami sedikit penurunan menjadi  $4.5 \times 10^{-9}$ .

- *Hadoop*: Nilai ROC pada input data 1 GB hingga 5 GB mengalami peningkatan yang drastis, mulai dari  $4.0 \times 10^{-8}$  menjadi  $5.1 \times 10^{-8}$ . Peningkatan ini berlanjut hingga 10 GB dengan nilai ROC mencapai  $5.3 \times 10^{-8}$ , sebelum menurun pada 15 GB menjadi  $4.8 \times 10^{-8}$ .

### Laju Perubahan *Throughput* terhadap Input Data (*Sort*)

Gambar 4.20 menunjukkan laju perubahan *throughput* Hadoop-Spark terhadap input data pada beban kerja *sort*. Hasil dan analisis dari gambar ini dapat dijelaskan sebagai berikut:

#### 1. 100 KB - 10 MB:

- *Spark*: Laju perubahan *throughput* pada data input 500 KB mencapai puncaknya sebesar  $2.8 \times 10^{-2}$ . Pada input data 5 MB dan 10 MB, laju perubahan *throughput* tetap stabil pada  $2.8 \times 10^{-2}$ .
- *Hadoop*: Laju perubahan *throughput* pada data input 500 KB mencapai puncaknya sebesar  $3.8 \times 10^{-2}$ . Pada input data 5 MB dan 10 MB, laju perubahan *throughput* tetap stabil pada  $3.8 \times 10^{-2}$ .

#### 2. 10 MB - 1 GB:

- *Spark*: Laju perubahan *throughput* pada data input 10 MB mencapai  $2.8 \times 10^{-2}$ . Setelah itu, laju perubahan *throughput* menurun secara bertahap dari 50 MB hingga 1 GB, mencapai nilai  $2.1 \times 10^{-2}$ .
- *Hadoop*: Pada rentang ini, laju perubahan *throughput* menunjukkan penurunan yang lebih tajam dari 10 MB ( $3.8 \times 10^{-2}$ ) hingga 1 GB ( $2.2 \times 10^{-2}$ ), dengan penurunan terbesar terjadi pada 500 MB ( $3.1 \times 10^{-2}$ ).

#### 3. 1 GB - 15 GB:

- *Spark*: Laju perubahan *throughput* pada rentang ini mengalami penurunan secara bertahap dari 1 GB ( $2.1 \times 10^{-2}$ ) hingga 15 GB ( $8.8 \times 10^{-4}$ ). Penurunan terbesar terjadi antara 5 GB ( $8.2 \times 10^{-3}$ ) dan 10 GB ( $2.9 \times 10^{-3}$ ).
- *Hadoop*: Laju perubahan *throughput* pada rentang ini juga mengalami penurunan yang signifikan dari 1 GB ( $2.2 \times 10^{-2}$ ) hingga 15 GB ( $2.5 \times 10^{-4}$ ). Penurunan terbesar terjadi antara 5 GB ( $3.9 \times 10^{-3}$ ) dan 10 GB ( $2.1 \times 10^{-4}$ ).

Analisis menunjukkan bahwa baik Spark maupun Hadoop menunjukkan tren penurunan laju perubahan *throughput* seiring dengan peningkatan ukuran input data.

Spark cenderung memiliki laju perubahan *throughput* yang lebih stabil pada ukuran input data kecil, sedangkan Hadoop menunjukkan penurunan yang lebih tajam terutama pada ukuran input data yang lebih besar.

### Laju Perubahan *Throughput* terhadap Input Data (*Word Count*)

Gambar 4.21 menunjukkan laju perubahan *throughput* Hadoop dan Spark terhadap input data pada beban kerja *word count*. Hasil dan analisis dari gambar ini dapat dijelaskan sebagai berikut:

#### 1. 100 KB - 10 MB:

- *Spark*: Laju perubahan *throughput* pada data input 500 KB mencapai puncaknya sebesar  $2.8 \times 10^{-2}$ . Pada input data 5 MB dan 10 MB, laju perubahan *throughput* tetap stabil pada  $2.8 \times 10^{-2}$ .
- *Hadoop*: Laju perubahan *throughput* pada data input 500 KB mencapai puncaknya sebesar  $3.7 \times 10^{-2}$ . Pada input data 5 MB, laju perubahan *throughput* sedikit menurun menjadi  $3.5 \times 10^{-2}$  dan tetap stabil pada input data 10 MB.

#### 2. 10 MB - 1 GB:

- *Spark*: Laju perubahan *throughput* pada data input 10 MB mencapai  $2.8 \times 10^{-2}$ . Setelah itu, laju perubahan *throughput* menurun secara bertahap dari 50 MB hingga 1 GB, mencapai nilai  $2.2 \times 10^{-2}$ .
- *Hadoop*: Pada rentang ini, laju perubahan *throughput* menunjukkan penurunan yang lebih tajam dari 10 MB ( $3.5 \times 10^{-2}$ ) hingga 1 GB ( $8.4 \times 10^{-3}$ ), dengan penurunan terbesar terjadi pada 500 MB ( $1.7 \times 10^{-2}$ ).

#### 3. 1 GB - 15 GB:

- *Spark*: Laju perubahan *throughput* pada rentang ini mengalami penurunan secara bertahap dari 1 GB ( $2.2 \times 10^{-2}$ ) hingga 15 GB ( $4.4 \times 10^{-3}$ ). Penurunan terbesar terjadi antara 5 GB ( $1.2 \times 10^{-2}$ ) dan 10 GB ( $5.9 \times 10^{-3}$ ).
- *Hadoop*: Laju perubahan *throughput* pada rentang ini juga mengalami penurunan yang signifikan dari 1 GB ( $8.4 \times 10^{-3}$ ) hingga 15 GB ( $1.3 \times 10^{-4}$ ). Penurunan terbesar terjadi antara 5 GB ( $8.9 \times 10^{-4}$ ) dan 10 GB ( $6.8 \times 10^{-5}$ ).

Analisis menunjukkan bahwa baik Spark maupun Hadoop menunjukkan tren penurunan laju perubahan *throughput* seiring dengan peningkatan ukuran input data. Spark cenderung memiliki laju perubahan throughput yang lebih stabil pada ukuran input data kecil, sedangkan Hadoop menunjukkan penurunan yang lebih tajam terutama pada ukuran input data yang lebih besar.

## Laju Perubahan Waktu Eksekusi Hadoop-Spark terhadap Input Data

Gambar 4.22 menunjukkan laju perubahan waktu eksekusi Hadoop dan Spark terhadap input data pada beban kerja *sort* dan *word count*. Hasil dan analisis dari gambar ini dapat dijelaskan sebagai berikut:

### 1. 100 KB - 10 MB:

- *Sort (Spark)*: Laju perubahan waktu eksekusi mencapai puncaknya pada 5 MB sebesar 5.3. Setelah itu, laju perubahan waktu eksekusi menurun drastis pada 10 MB menjadi  $-5.0$ .
- *Word Count (Hadoop)*: Laju perubahan waktu eksekusi mengalami puncak pada 1 MB sebesar 9.8. Setelah itu, terjadi penurunan yang signifikan pada 500 KB menjadi  $-3.7$  dan terus menurun hingga 10 MB menjadi  $5.6 \times 10^{-2}$ .

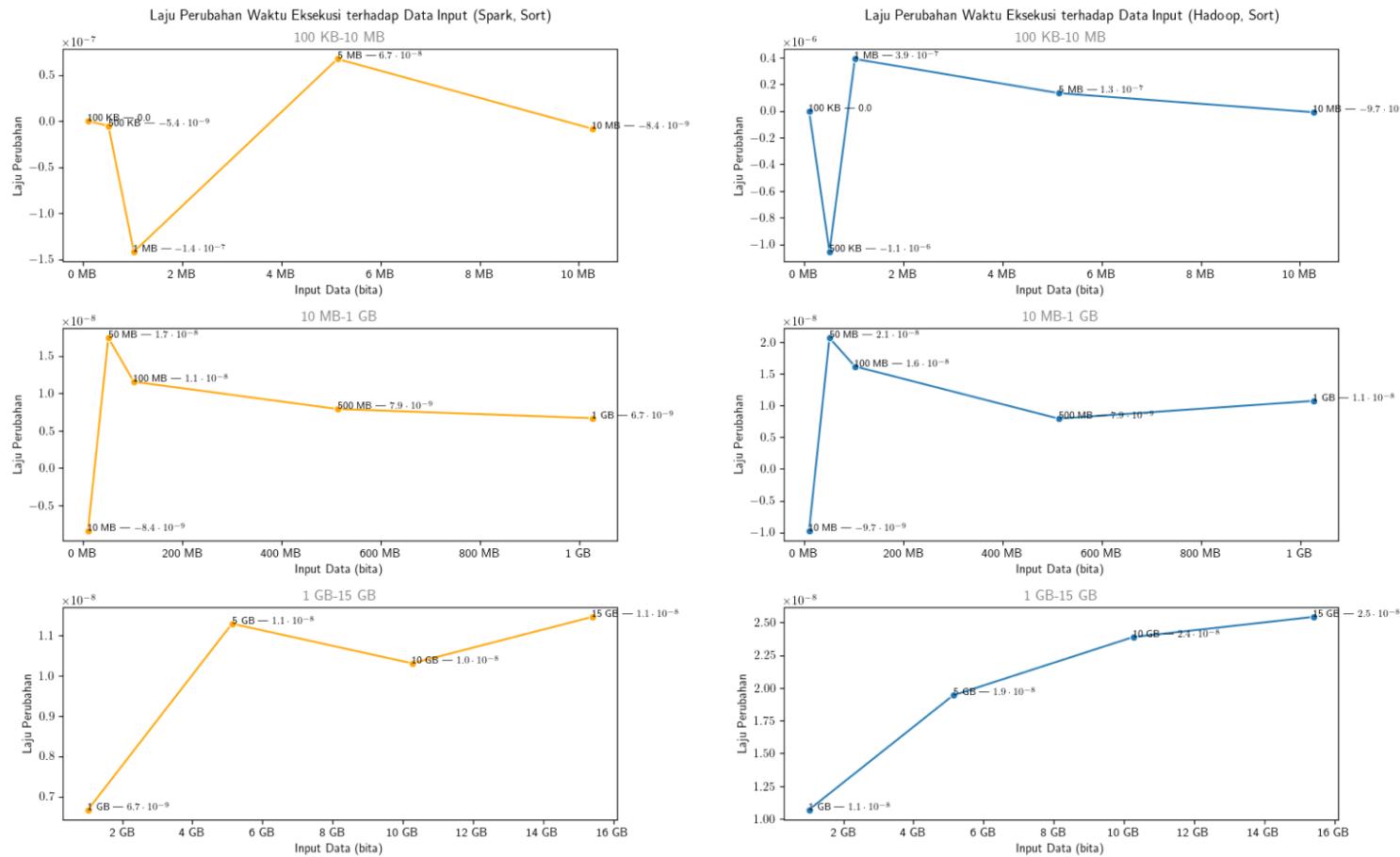
### 2. 10 MB - 1 GB:

- *Sort (Spark)*: Laju perubahan waktu eksekusi menunjukkan peningkatan signifikan dari 50 MB ( $-5.0 \times 10^{-2}$ ) hingga 500 MB ( $3.2 \times 10^9$ ), lalu meningkat lagi hingga 1 GB mencapai  $5.5 \times 10^9$ .
- *Word Count (Hadoop)*: Laju perubahan waktu eksekusi meningkat tajam dari 50 MB ( $1.0 \times 10^9$ ) hingga 1 GB ( $2.0 \times 10^1$ ).

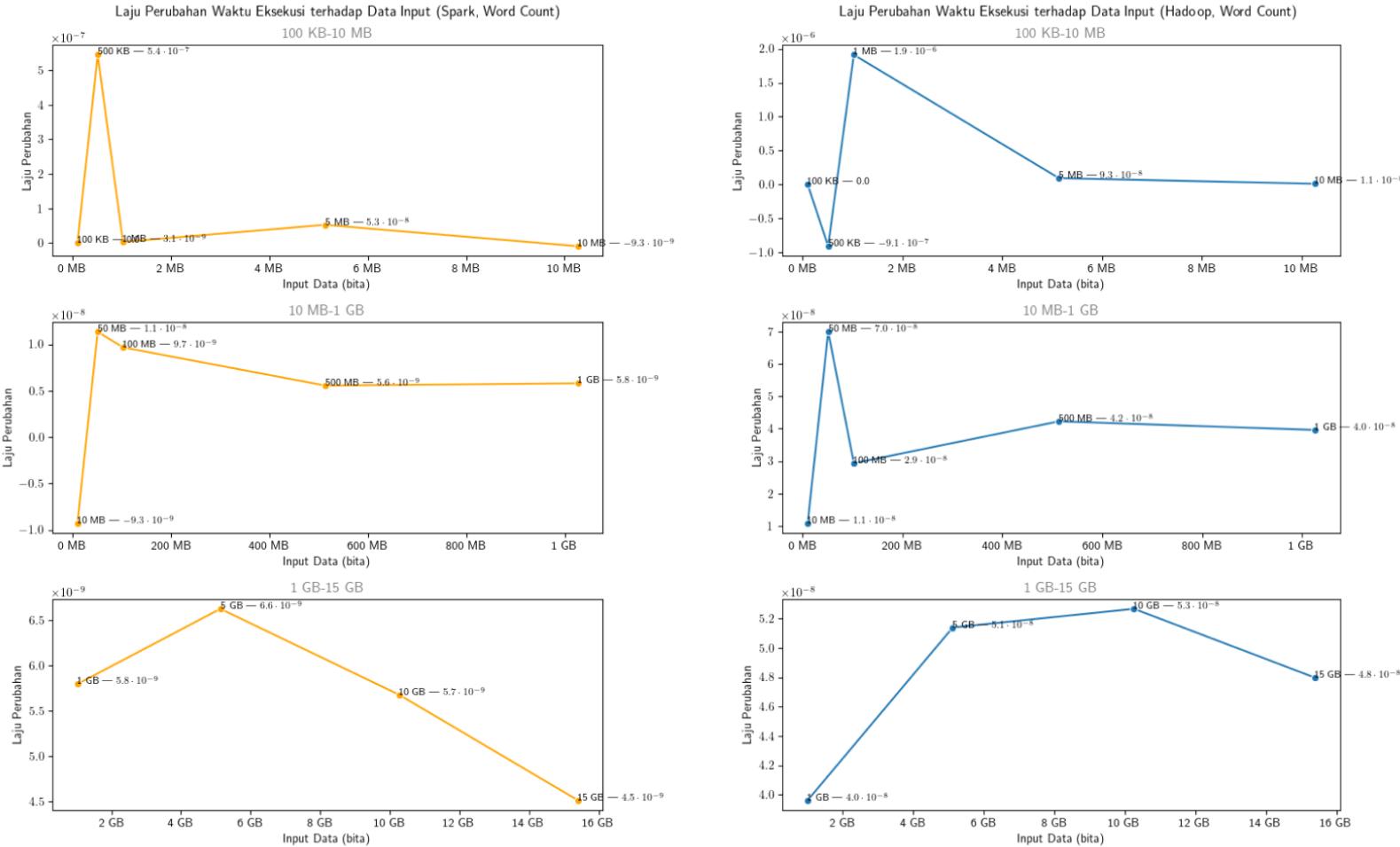
### 3. 1 GB - 15 GB:

- *Sort (Spark)*: Laju perubahan waktu eksekusi terus meningkat seiring dengan penambahan ukuran input data. Pada 5 GB, laju perubahan mencapai  $8.0 \times 10^1$ , lalu meningkat signifikan pada 10 GB mencapai  $1.2 \times 10^2$  dan mencapai puncaknya pada 15 GB sebesar  $1.3 \times 10^2$ .
- *Word Count (Hadoop)*: Laju perubahan waktu eksekusi juga meningkat seiring dengan penambahan ukuran input data. Pada 5 GB, laju perubahan mencapai  $2.1 \times 10^2$ , lalu meningkat signifikan pada 10 GB mencapai  $2.7 \times 10^2$  dan sedikit menurun pada 15 GB menjadi  $2.5 \times 10^2$ .

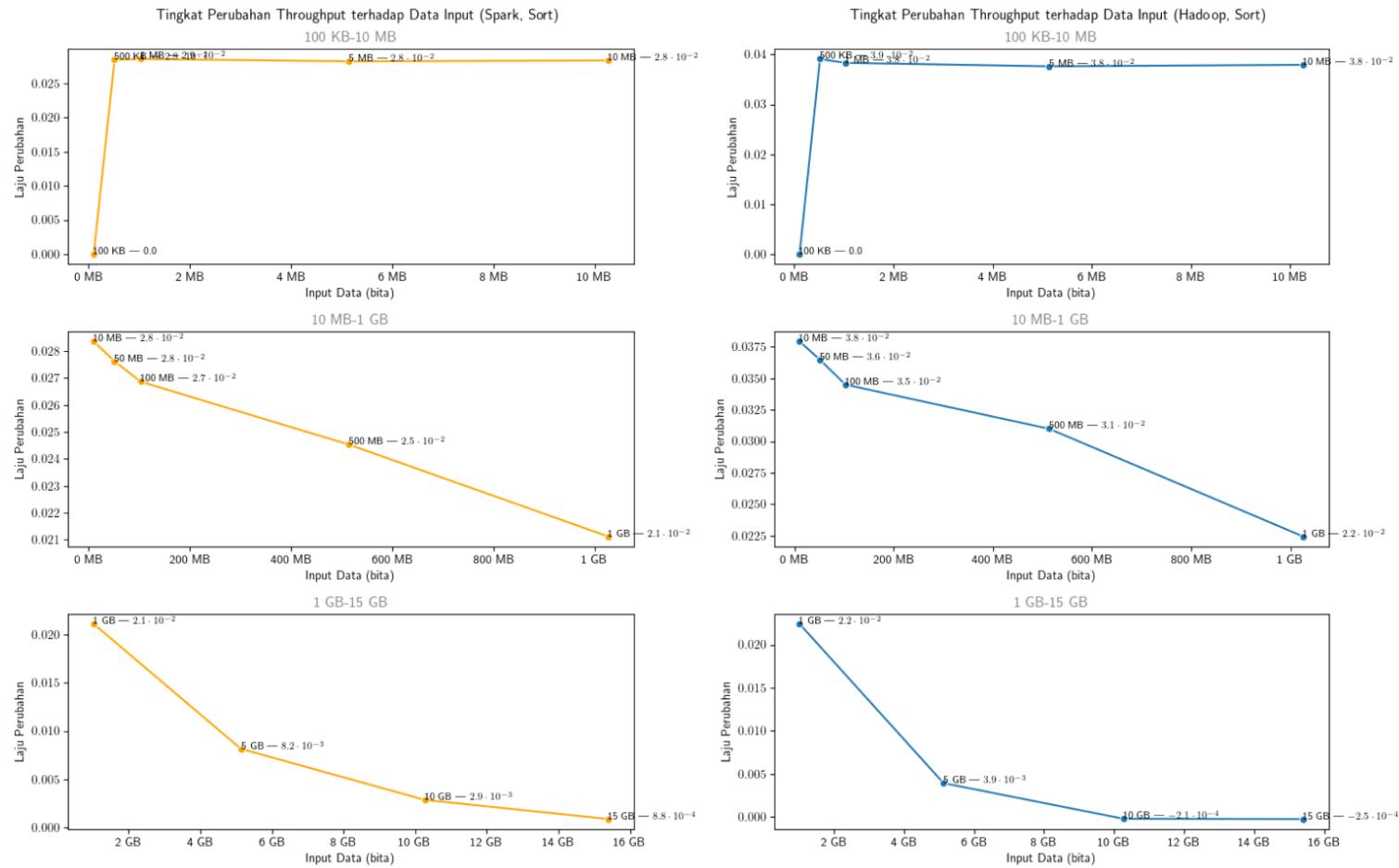
Analisis menunjukkan bahwa baik Spark maupun Hadoop mengalami peningkatan laju perubahan waktu eksekusi seiring dengan peningkatan ukuran input data. Spark menunjukkan peningkatan yang lebih stabil pada beban kerja *sort*, sedangkan Hadoop menunjukkan peningkatan yang lebih fluktuatif pada beban kerja *word count*. Penurunan drastis pada beberapa titik input data tertentu menunjukkan adanya inefisiensi atau bottleneck dalam pengolahan data pada ukuran tersebut.



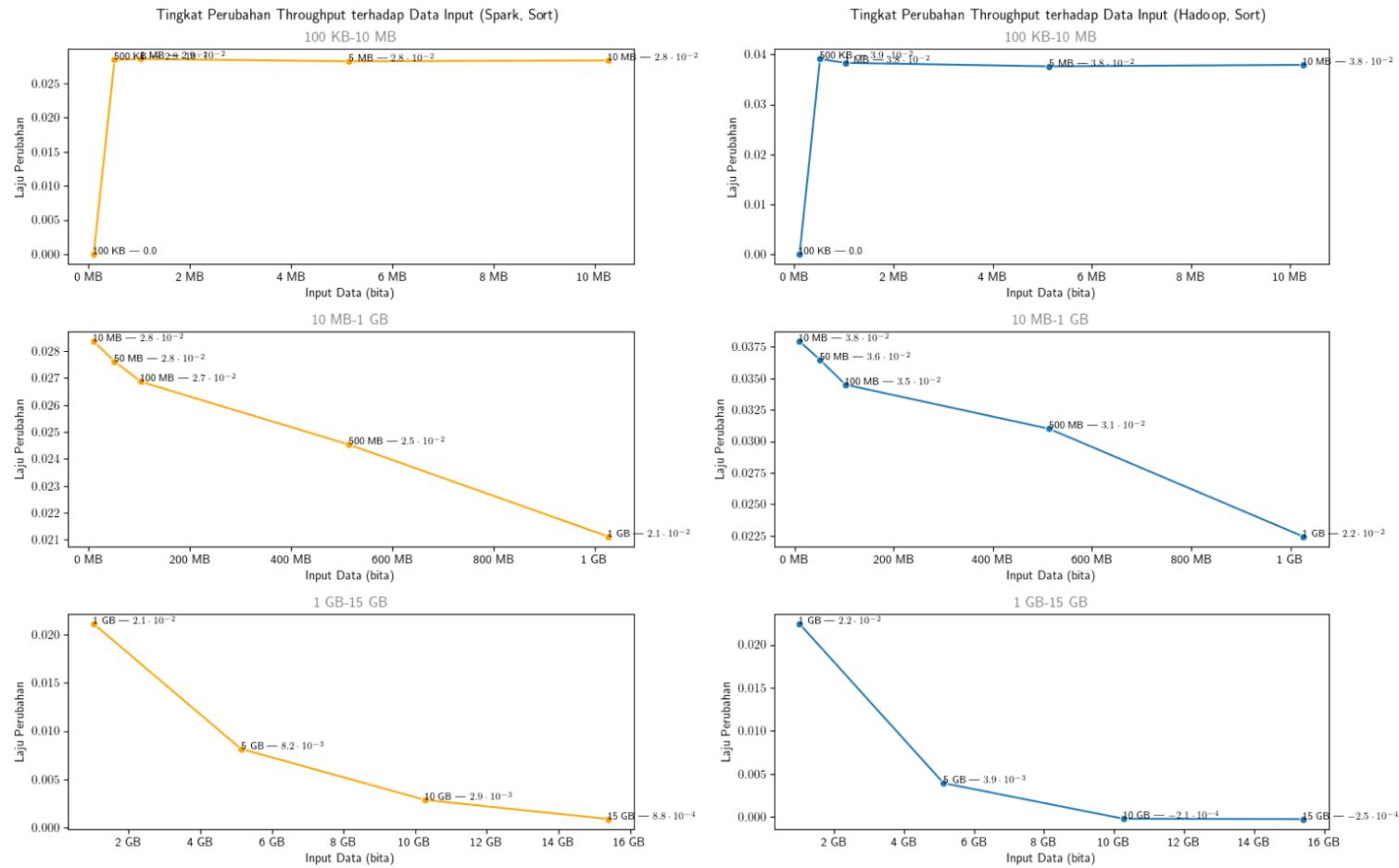
Gambar 4.18 Laju Perubahan Waktu Eksekusi terhadap Input Data (*Sort*)



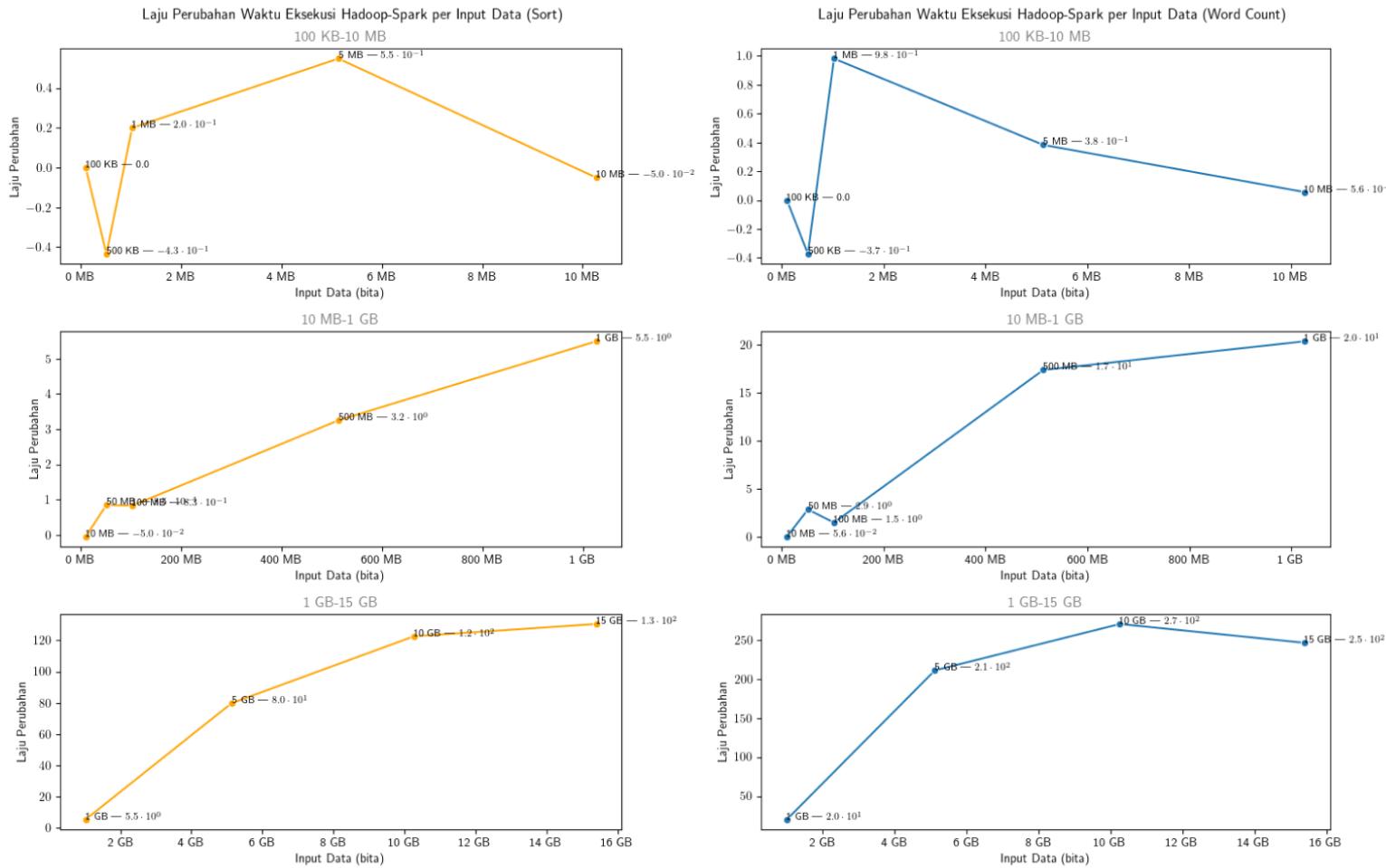
Gambar 4.19 Laju Perubahan Waktu Eksekusi terhadap Input Data (*Word Count*)



Gambar 4.20 Laju Perubahan *Throughput* terhadap Input Data (*Sort*)



Gambar 4.21 Laju Perubahan *Throughput* terhadap Input Data (*Word Count*)



**Gambar 4.22** Laju Perubahan Waktu Eksekusi Hadoop-Spark terhadap Input Data

## **4.6 Analisis dan Evaluasi Hasil Eksperimen: Penggunaan Sumber Daya**

### **4.6.1 Penggunaan CPU**

Gambar 4.23 dan 4.24 menunjukkan pola penggunaan CPU oleh Hadoop dan Spark untuk beban kerja *sort* dan *word count* pada berbagai ukuran data. Sumbu x mewakili waktu dalam detik, sedangkan sumbu y mewakili persentase penggunaan CPU. Setiap grafik menunjukkan ukuran data yang berbeda, mulai dari 100 KB hingga 15 GB. Titik hitam menandakan titik perpotongan Hadoop dan Spark.

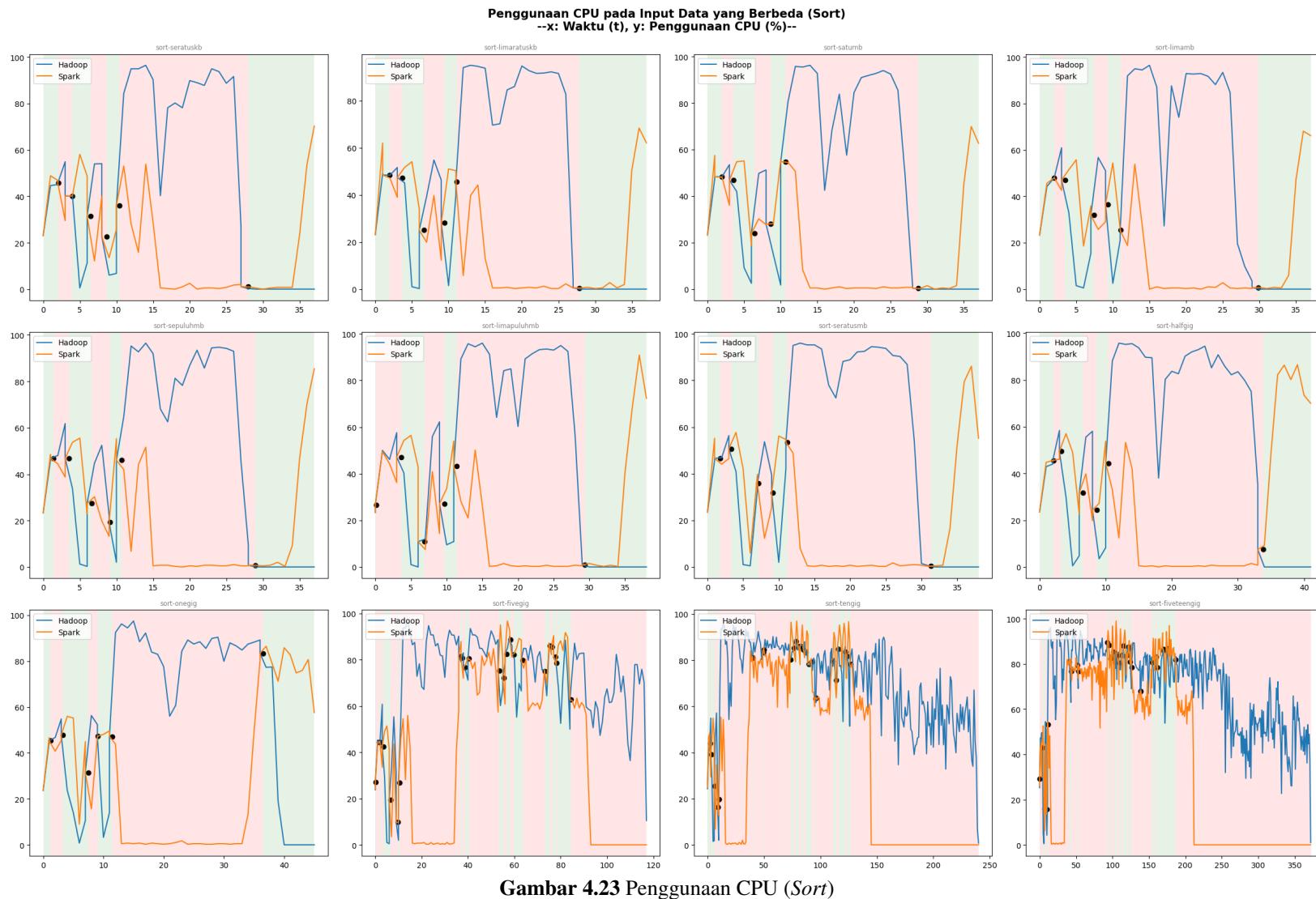
Pada beban kerja *sort* (Gambar 4.23), perbedaan pola penggunaan CPU antara Hadoop dan Spark semakin terlihat pada ukuran data yang lebih besar. Jika dilihat pada input data 100 KB-1 GB, penggunaan CPU pada masing-masing Hadoop dan Spark tidak berbeda jauh. Selanjutnya, pada input data 5 GB, 10 GB, dan 15 GB, penggunaan CPU Hadoop sangat fluktuatif berkisar pada 60%-95% pada dua per tiga bagian waktu awal eksekusi, dan pada satu per tiga waktu eksekusi mengalami penurunan yang berkisar pada 20%-80%. Berbeda dengan Hadoop, Spark memiliki penggunaan CPU yang lebih stabil (terkadang pada 60%-80%, dan terkadang pada 80%-100%).

Pada beban kerja *word count* (Gambar 4.24), Hadoop menunjukkan penggunaan CPU yang lebih fluktuatif (naik turun) dan tinggi secara keseluruhan dibandingkan dengan Spark. Penggunaan CPU pada 10 detik pertama pada Hadoop berkisar pada 10%-60%. Setelah itu, penggunaan CPU-nya naik sampai ke 100% dan naik turun. Jika dilihat dari input data yang lebih besar (15 GB), Hadoop memiliki pola CPU yang hampir sama pada setiap data input, yaitu 10 detik pertama berada pada 10%-60%, setengah pertama naik turun pada penggunaan CPU 60%-100%, dan setengah terakhir menurunkan penggunaan CPU pada 50%-90%. Selanjutnya, Spark juga memiliki pola tersendiri. Penggunaan CPU Spark pada detik pertama sampai detik ke 15 fluktuatif pada 20%-50%, hingga pada detik 16 sampai detik ke 35 turun ke 0%, dan baru naik lagi pada detik ke 35. Hal yang menarik juga adalah penggunaan CPU Spark tidak menyentuh 100%, tetapi memiliki waktu eksekusi beban kerja yang lebih cepat pada *word count*.

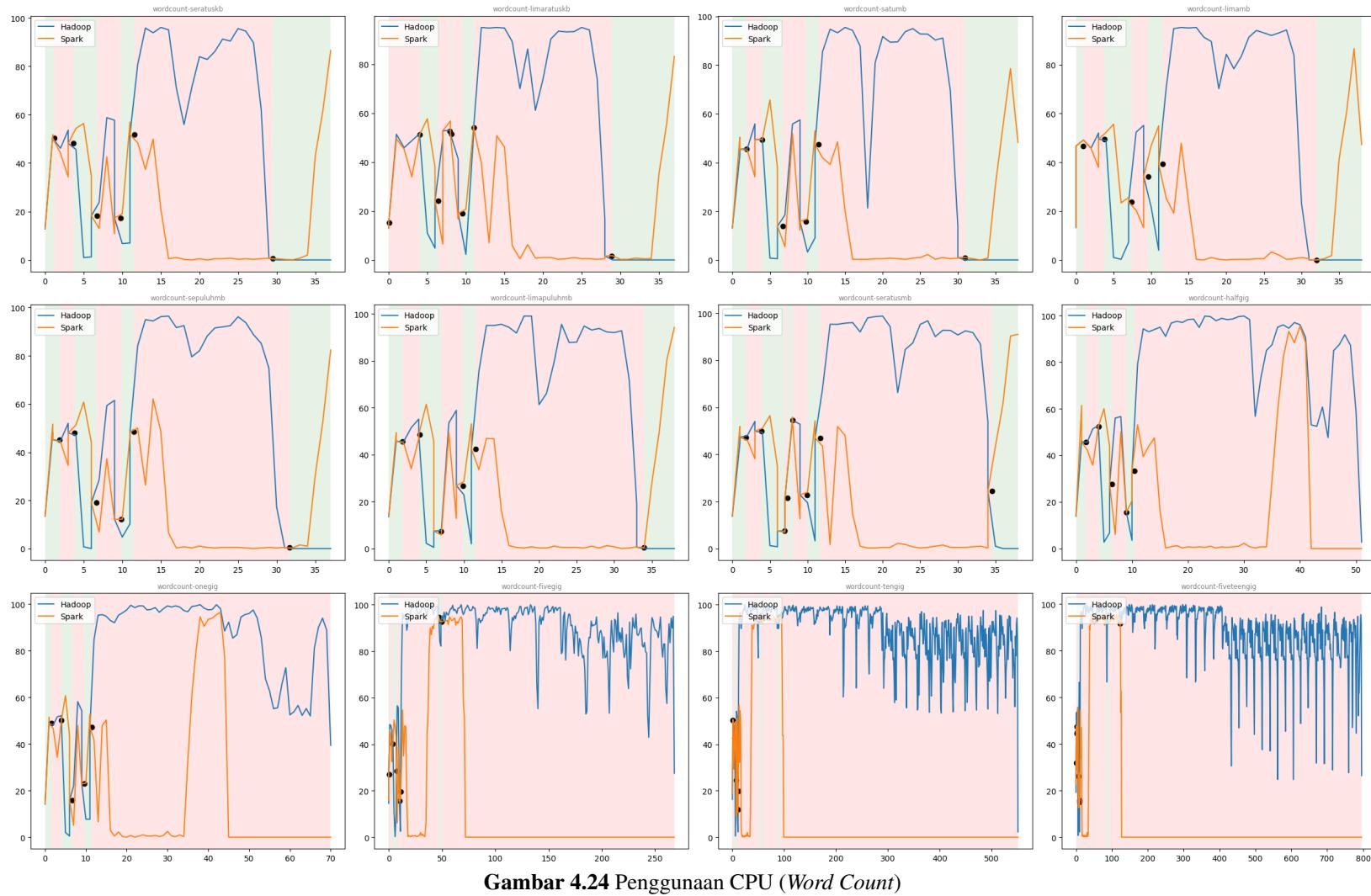
Pada kedua tugas, terlihat bahwa beban kerja *word count* cenderung menunjukkan pola penggunaan CPU yang lebih tinggi dan konsisten dibandingkan dengan beban kerja *sort*. Pada beban kerja *word count*, umumnya penggunaan CPU Hadoop lebih tinggi dan merata di sepanjang waktu eksekusi jika dibandingkan dengan Spark. Di sisi lain, beban kerja *sort* menunjukkan penggunaan CPU yang cenderung lebih fluktuatif, dengan periode lonjakan dan penurunan yang signifikan, baik pada Hadoop maupun Spark.

Secara keseluruhan, Hadoop cenderung menggunakan CPU dengan lebih intensif

dan fluktuatif dibandingkan Spark, terutama pada tugas *word count*. Spark, meskipun tidak selalu menggunakan CPU hingga 100%, mampu menyelesaikan tugas dengan waktu eksekusi yang lebih cepat, menunjukkan efisiensi penggunaan sumber daya yang lebih baik.

**Gambar 4.23 Penggunaan CPU (Sort)**

Penggunaan CPU pada Input Data yang Berbeda (Word Count)  
--x: Waktu (t), y: Penggunaan CPU (%)--



Gambar 4.24 Penggunaan CPU (Word Count)

Gambar 4.25 dan 4.26 menunjukkan perbandingan *state* penggunaan CPU antara Hadoop ( $U_h$ ) dan Spark ( $U_s$ ) pada beban kerja *sort* dan *word count*. *State*  $U_h > U_s$  menunjukkan durasi waktu di mana penggunaan CPU Hadoop lebih tinggi daripada Spark, sementara *state*  $U_h < U_s$  menunjukkan sebaliknya.

Pada Gambar 4.25, terlihat bahwa durasi *state*  $U_h > U_s$  secara konsisten lebih tinggi dibandingkan dengan  $U_h < U_s$  pada semua ukuran data. Ini menunjukkan bahwa Hadoop biasanya menggunakan lebih banyak CPU dibandingkan Spark untuk beban kerja *sort*.

Contoh yang menonjol adalah pada ukuran data:

1. 100 KB:  $U_h > U_s$  selama 21.7 detik, sedangkan  $U_h < U_s$  hanya 15.3 detik.
2. 1 GB:  $U_h > U_s$  selama 28.71 detik, sedangkan  $U_h < U_s$  hanya 16.29 detik.
3. 10 GB:  $U_h > U_s$  selama 207.37 detik, sedangkan  $U_h < U_s$  hanya 32.63 detik.
4. 15 GB:  $U_h > U_s$  selama 314.86 detik, sedangkan  $U_h < U_s$  hanya 57.14 detik.

Ketika ukuran data meningkat, perbedaan antara  $U_h > U_s$  dan  $U_h < U_s$  semakin besar. Hal ini menunjukkan bahwa Hadoop semakin intensif dalam penggunaan CPU dibandingkan Spark seiring dengan bertambahnya ukuran data.

Pada Gambar 4.26, variasi dalam perbandingan *state* lebih terlihat dibandingkan beban kerja *sort*. Namun, secara umum, durasi *state*  $U_h > U_s$  tetap lebih tinggi dibandingkan  $U_h < U_s$ .

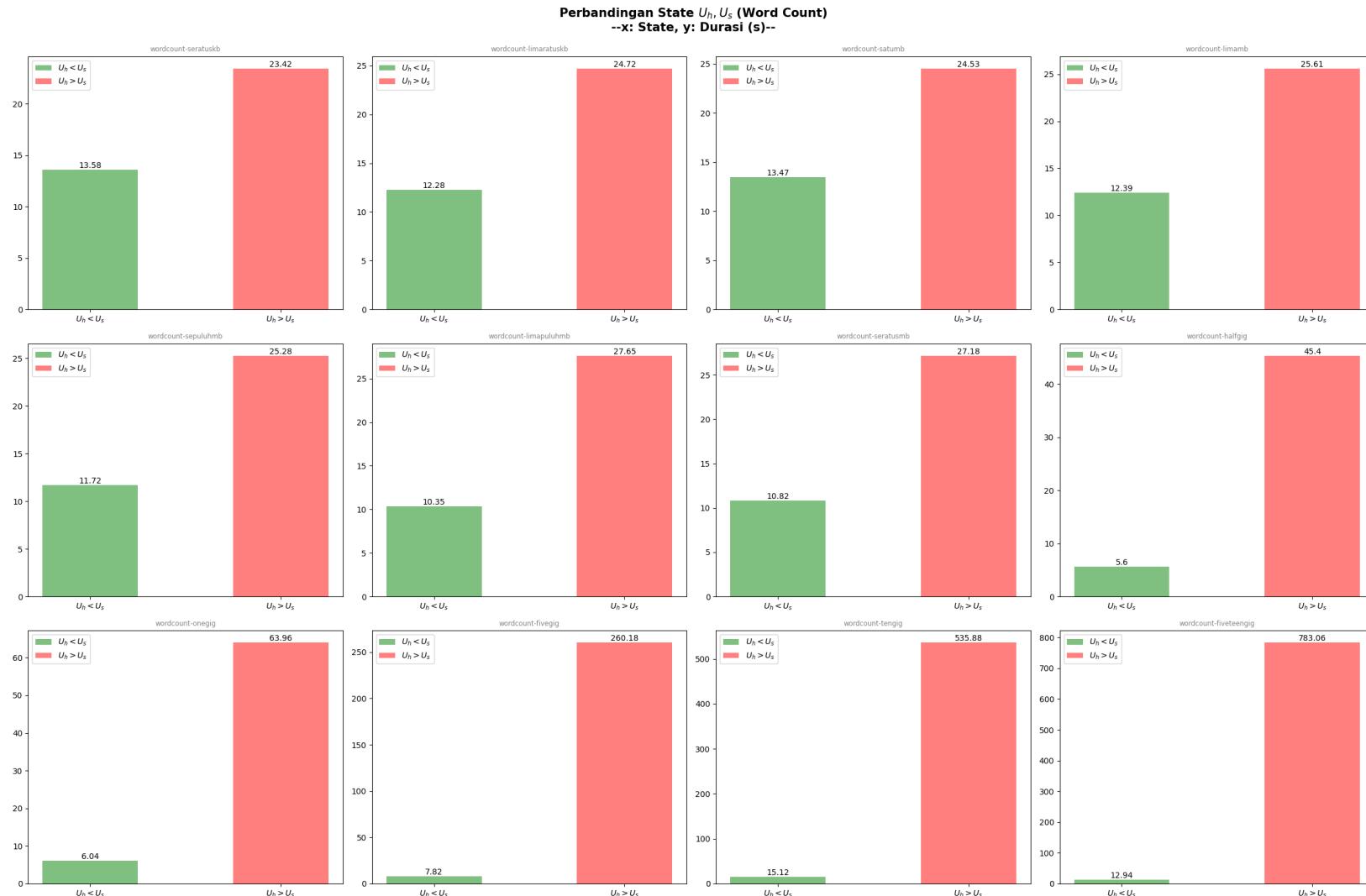
Contoh yang menonjol adalah pada ukuran data:

1. 100 KB:  $U_h > U_s$  selama 23.42 detik, sedangkan  $U_h < U_s$  hanya 13.58 detik.
2. 1 GB:  $U_h > U_s$  selama 63.96 detik, sedangkan  $U_h < U_s$  hanya 6.04 detik.
3. 10 GB:  $U_h > U_s$  selama 535.88 detik, sedangkan  $U_h < U_s$  hanya 15.12 detik.
4. 15 GB:  $U_h > U_s$  selama 783.06 detik, sedangkan  $U_h < U_s$  hanya 12.94 detik.

Pada beban kerja *word count*, durasi  $U_h < U_s$  tidak menunjukkan pola yang konsisten dengan ukuran data, tetapi durasi  $U_h > U_s$  cenderung meningkat seiring dengan bertambahnya ukuran data. Ini menunjukkan bahwa Hadoop lebih sering mengalami penggunaan CPU yang lebih tinggi dibandingkan Spark, terutama pada ukuran data yang lebih besar.

Berdasarkan analisis sebelumnya, Hadoop cenderung menggunakan CPU lebih intensif dibandingkan Spark pada kedua jenis beban kerja (*sort* dan *word count*). Perbedaan ini semakin signifikan seiring dengan bertambahnya ukuran data, terutama pada *sort*. Pada *word count*, meskipun terdapat variasi pada durasi  $U_h < U_s$ , durasi  $U_h > U_s$  tetap menunjukkan dominasi penggunaan CPU oleh Hadoop.

**Gambar 4.25 Perbandingan State (Sort)**

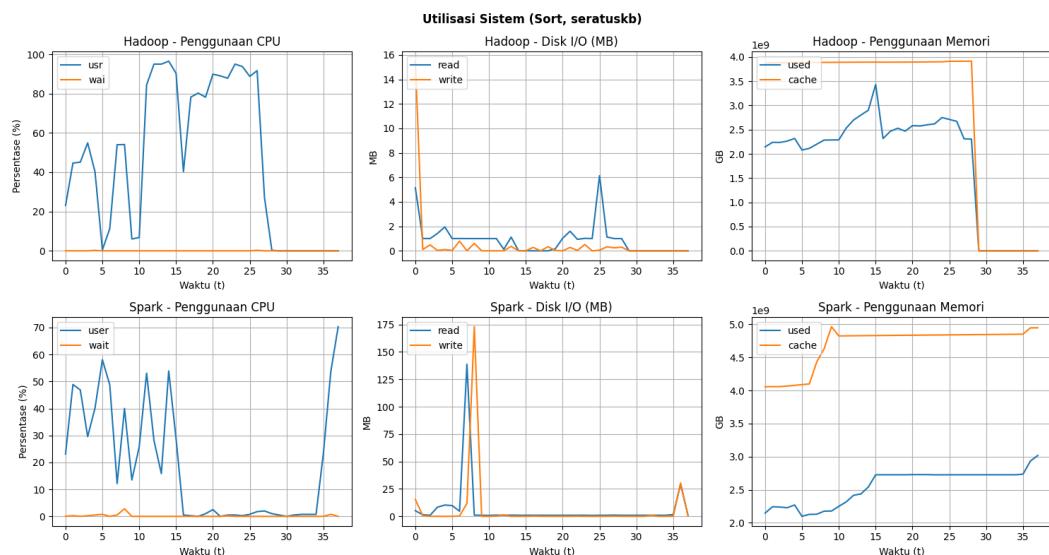
**Gambar 4.26 Perbandingan State (Word Count)**

#### 4.6.2 Utilisasi Sistem

Utilisasi sistem secara lengkap dapat dilihat pada Lampiran G untuk beban kerja *sort* dan Lampiran H untuk beban kerja *word count*. Setiap gambar akan terdiri dari dua baris dan tiga kolom. Baris pertama berisi visualisasi utilisasi sistem untuk Hadoop dan baris kedua berisi visualisasi untuk Spark. Setiap baris berisi tiga utilisasi sistem, yaitu

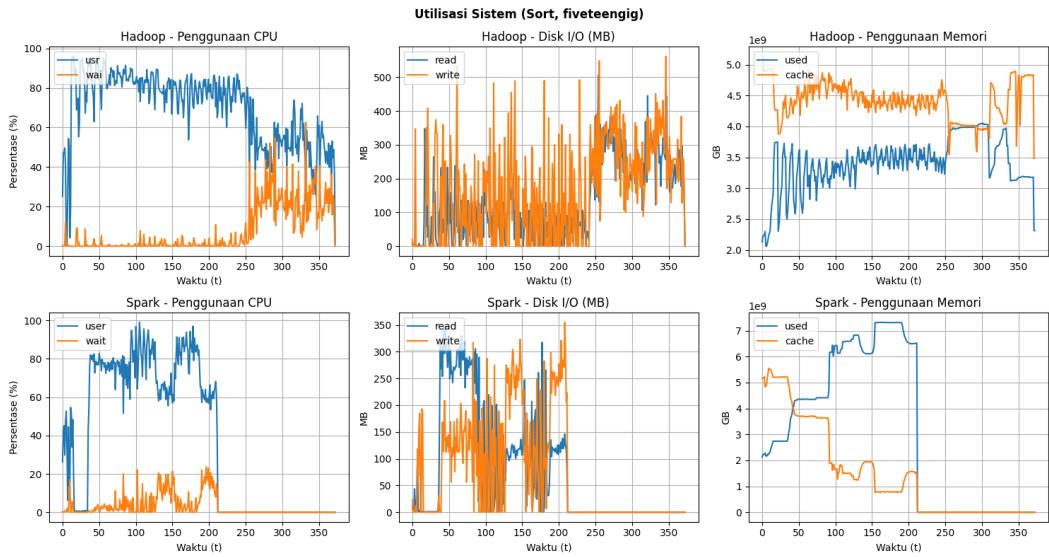
1. Penggunaan CPU (%)
2. *Disk I/O* (MB)
3. Memori (GB)

Berdasarkan analisis pola penggunaan CPU pada tahap sebelumnya yang menunjukkan bahwa penggunaan CPU Hadoop dan Spark memiliki polanya masing-masing, maka pada tahap ini hanya akan ditampilkan utilisasi sistem pada input data terkecil (100 KB) dan input data terbesar (15 GB).



Gambar 4.27 Utilisasi Sistem (*Sort*) pada Input Data 100 KB

Pada beban kerja *sort* dan input data sebesar 100 KB (seperti yang ditunjukkan Gambar 4.27), Hadoop memiliki penggunaan CPU yang lebih tinggi, yaitu hampir menyentuh 100%. Sedangkan, pada Spark, penggunaan CPU tertinggi sekitar 70%. Selanjutnya, ditinjau dari *disk I/O*, aktivitas baca (*read*) dan tulis (*write*) pada Hadoop cenderung lebih sering namun memiliki kecepatan yang lebih lambat dibandingkan dengan Spark. Spark memiliki aktivitas baca tulis yang lebih tinggi, yaitu 140 MB untuk baca dan 175 MB untuk tulis. Kemudian, jika ditinjau dari penggunaan memori, Hadoop dan Spark memiliki penggunaan memori yang tidak berbeda jauh, yaitu sekitar 2 GB-3.5 GB.

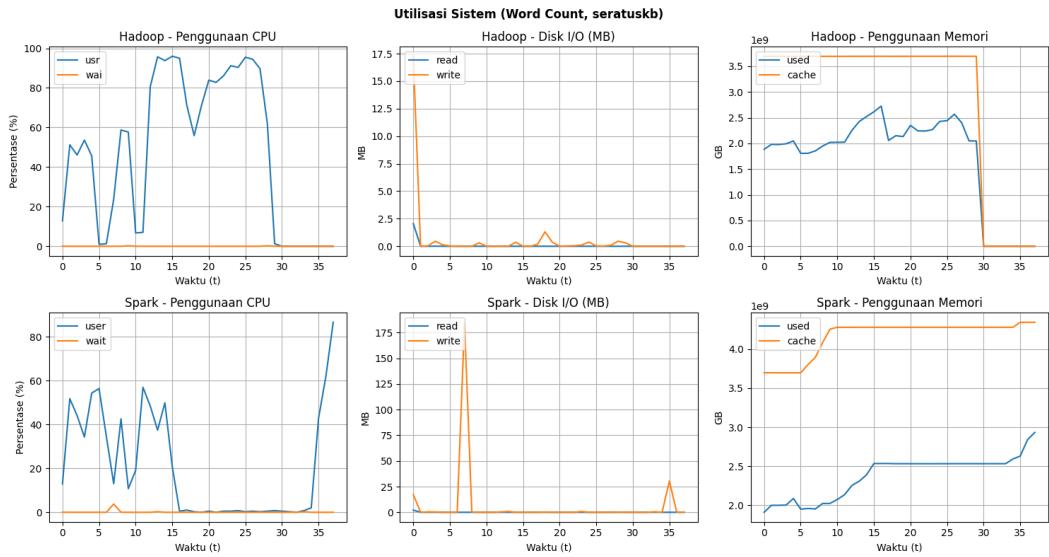


Gambar 4.28 Utilisasi Sistem (*Sort*) pada Input Data 15 GB

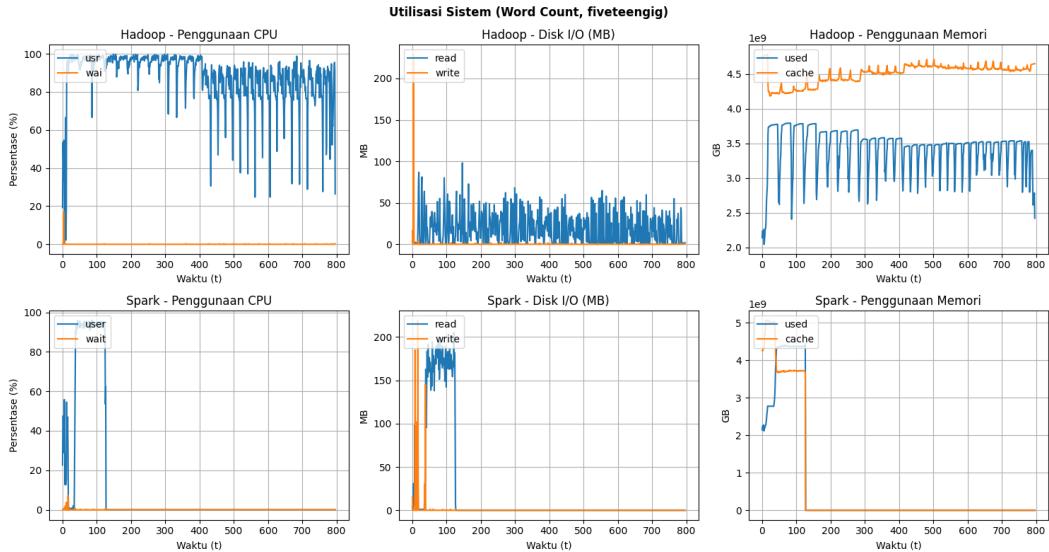
Pada beban kerja *sort* dan input data yang lebih besar, yaitu 15 GB (seperti yang ditunjukkan pada Gambar 4.28), utilisasi sistem pada Hadoop dan Spark lebih terlihat jelas perbedaannya. Jika dilihat pada penggunaan CPU, Hadoop membutuhkan waktu penggunaan CPU yang lebih lama, yaitu sekitar 370 detik, dimana Spark hanya membutuhkan waktu sekitar 220 detik. Selanjutnya, Hadoop memiliki penggunaan CPU "user" yang lebih stabil, jika dibandingkan dengan Spark yang naik turun secara konstan. Penggunaan CPU "wait" akan naik ketika penggunaan CPU "user" itu turun. Bergeser pada *disk I/O*, Hadoop memiliki siklus baca tulis yang lebih intensif jika dibandingkan pada Spark. Selanjutnya, jika dilihat dari penggunaan memori, Spark lebih "rakus" akan memori. Hal ini ditandai dengan puncaknya membutuhkan memori sekitar 6-7 GB, dimana Hadoop hanya membutuhkan memori sekitar 3.5 GB.

Selanjutnya beban kerja *word count*. Pada beban kerja *word count* dengan input data 100 KB, pola yang didapatkan untuk penggunaan CPU, *disk I/O*, dan penggunaan memori tidak berbeda jauh dengan beban kerja *sort* dengan ukuran input data yang sama.

Pada beban kerja *word count* dan input data yang lebih besar, yaitu 15 GB (seperti yang ditunjukkan pada Gambar 4.30), utilisasi sistem pada Hadoop dan Spark lebih terlihat jelas perbedaannya. Pada penggunaan CPU, Hadoop memiliki aktivitas penggunaan yang lebih tinggi dan lebih konstan sampai akhir waktu eksekusi. Hal ini berbeda dengan Spark yang hanya butuh waktu sekitar 150 detik saja dengan penggunaan CPU yang hanya 90%. Selanjutnya, pada aktivitas baca tulis (*disk I/O*), Hadoop memiliki aktivitas baca yang lebih intensif (sepanjang waktu eksekusi) dengan sedikit aktivitas tulis (pada awal waktu eksekusi). Aktivitas baca yang



Gambar 4.29 Utilisasi Sistem (*Word Count*) pada Input Data 100 KB



Gambar 4.30 Utilisasi Sistem (*Word Count*) pada Input Data 15 GB

dilakukan oleh Hadoop berada pada ukuran 50-100 MB setiap waktunya. Pada Spark, aktivitas baca tersebut berukuran 150-200 MB. Kemudian, jika ditinjau melalui penggunaan memori, memori yang dibutuhkan Hadoop berkisar pada 2.5-3.7 GB. Pada Spark, memori yang dibutuhkan berkisar pada 2-4.5 GB.

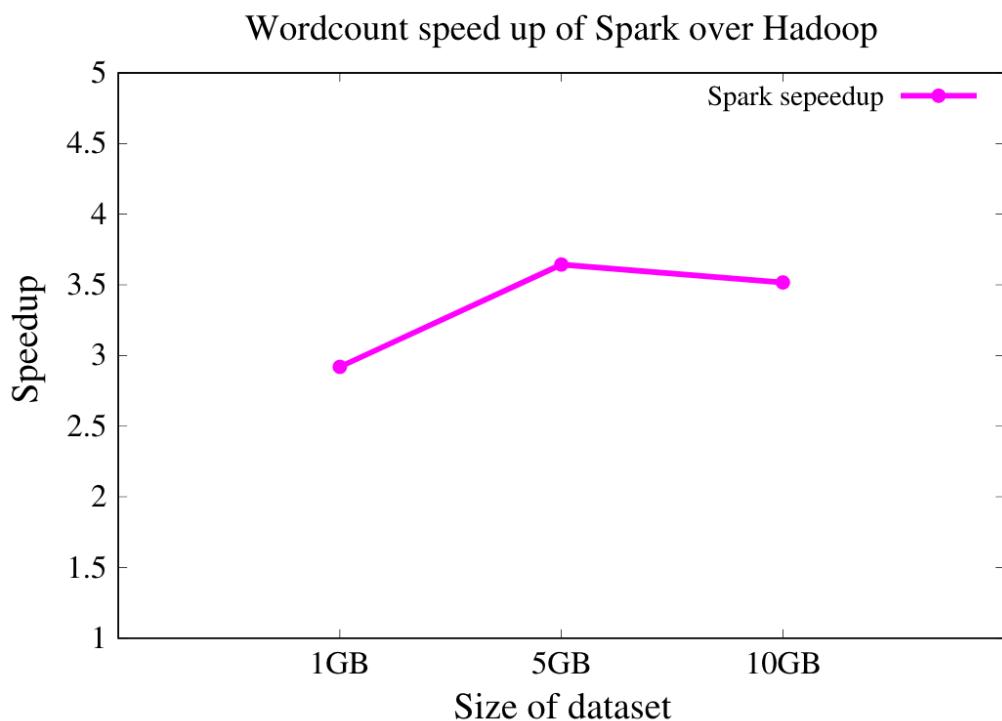
Hadoop menunjukkan aktivitas *Disk I/O* yang jauh lebih tinggi dibandingkan dengan Spark, terutama pada beban kerja *sort*. Grafik *Disk I/O* Hadoop menunjukkan lonjakan aktivitas baca dan tulis yang signifikan sepanjang waktu eksekusi. Hal ini sesuai dengan pendekatan berbasis disk Hadoop yang membutuhkan pembacaan dan penulisan data ke *disk* secara intensif. Sebaliknya, Spark, dengan arsitektur in-memory, meminimalkan operasi *Disk I/O*. Grafik *Disk I/O* Spark menunjukkan

aktivitas yang jauh lebih rendah dan stabil, yang berkontribusi pada peningkatan performanya.

Spark menunjukkan penggunaan memori yang lebih tinggi dan stabil dibandingkan dengan Hadoop, terutama pada beban kerja *sort*. Grafik penggunaan memori Spark menunjukkan garis yang cenderung mendatar pada tingkat utilisasi yang tinggi, menunjukkan bahwa Spark menyimpan data dalam RAM untuk akses yang lebih cepat dan pemrosesan yang efisien. Penggunaan memori Hadoop lebih rendah dan fluktuatif, menunjukkan bahwa Hadoop tidak memanfaatkan memori secara optimal. Analisis pemantauan sistem menegaskan keunggulan Spark dalam hal efisiensi dan optimasi penggunaan sumber daya komputasi dibandingkan dengan Hadoop. Spark mampu memaksimalkan penggunaan CPU, meminimalkan operasi *Disk I/O*, dan memanfaatkan memori secara efisien, yang berkontribusi pada performa dan skalabilitas yang lebih baik dalam tugas-tugas pemrosesan data besar.

#### 4.7 Perbandingan dengan Penelitian Sebelumnya

Penelitian terdahulu yang melakukan perbandingan Hadoop dan Spark menggunakan HiBench telah mendapatkan hasil seperti pada Gambar 4.31. Pada gambar tersebut terlihat bahwa pada input data 1 GB dan pada beban kerja *word count*, terjadi peningkatan performa sebesar tiga kali lipat, pada input data 5 GB sebesar 3.5 lipat, dan begitu juga pada input data 10 GB.



Gambar 4.31 Rasio Peningkatan Performa Spark-Hadoop Berdasarkan Input Data [1]

Penelitian ini menghasilkan hasil yang serupa, seperti yang ditunjukkan pada Tabel 4.3. Pada penelitian ini, rasio peningkatan performa naik secara bertahap dari input data 1 GB, 5 GB, 10 GB, hingga 15 GB. Hal ini dapat terjadi karena adanya perbedaan konfigurasi perangkat keras dan perbedaan versi perangkat lunak yang digunakan.

**Tabel 4.3** Rasio Peningkatan Performa Spark-Hadoop Berdasarkan Input Data (Baru)

Aplikasi	Input Data	Rata-rata Waktu Eksekusi (s)	Rasio Peningkatan
Spark	1 GB	42,164	1,67
Hadoop	1 GB	70,37	
Spark	5 GB	69,36	4,06
Hadoop	5 GB	281,3564	
Spark	10 GB	98,5008	5,6
Hadoop	10 GB	551,6242	
Spark	15 GB	121,6664	6,56
Hadoop	15 GB	797,8058	

Pada Tabel 4.3, dapat dilihat bahwa Spark menunjukkan peningkatan performa yang signifikan dibandingkan Hadoop pada berbagai ukuran input data. Pada input data 1 GB, Spark memiliki rasio peningkatan sebesar 1.67 kali lipat dibandingkan Hadoop. Pada input data 5 GB, rasio peningkatan naik menjadi 4.06 kali lipat. Begitu juga dengan input data 10 GB dan 15 GB, rasio peningkatannya masing-masing sebesar 5.6 kali lipat dan 6.56 kali lipat.

Secara keseluruhan, hasil penelitian ini konsisten dengan penelitian sebelumnya, namun menunjukkan peningkatan performa yang lebih signifikan pada ukuran data yang lebih besar. Hal ini menunjukkan bahwa Spark lebih efisien dalam menangani data dalam skala besar dibandingkan Hadoop, terutama pada beban kerja *word count*.

## BAB V

### PENUTUP

#### 5.1 Kesimpulan

Penelitian ini mengevaluasi dan membandingkan kinerja dua *platform* komputasi terdistribusi populer, Hadoop dan Spark, dalam mengolah data teks khususnya penerapan algoritma *sort* dan *word count* pada *platform cloud* DigitalOcean.

Berdasarkan hasil penelitian, Spark menunjukkan performa yang lebih unggul dibandingkan Hadoop dalam sebagian besar skenario. Spark mampu menyelesaikan tugas *sort* dan *word count* dengan waktu eksekusi yang lebih cepat dan *throughput* yang lebih tinggi, terutama pada data berukuran besar (di atas 500 MB untuk *word count* dan 5 GB untuk *sort*). Hal ini disebabkan arsitektur *in-memory* Spark yang memungkinkan pemrosesan data lebih cepat dengan meminimalkan akses ke *disk*. Analisis penggunaan sumber daya menunjukkan bahwa Spark lebih efisien dalam memanfaatkan CPU dan memori, serta meminimalkan operasi *disk I/O* dibandingkan Hadoop. Hal ini berkontribusi pada performa dan skalabilitas Spark yang lebih baik dalam menangani data besar.

Hasil penelitian ini menegaskan bahwa Spark merupakan pilihan yang lebih tepat untuk pemrosesan data besar dibandingkan Hadoop, terutama jika *throughput* dan waktu eksekusi menjadi pertimbangan utama.

#### 5.2 Saran

Penelitian selanjutnya dapat memfokuskan dengan menguji performa Hadoop dan Spark pada algoritma *machine learning*, menganalisis pengaruh penambahan jumlah *node* terhadap skalabilitas, serta membangun model *forecasting* performa berdasarkan *rate of change* untuk memprediksi waktu eksekusi dan *throughput* pada beban kerja yang lebih besar. Dengan menjalankan penelitian lanjutan yang menggabungkan aspek-aspek tersebut, pemahaman yang lebih komprehensif mengenai keunggulan dan kelemahan Hadoop dan Spark dapat diperoleh, sehingga memungkinkan pemilihan platform yang optimal berdasarkan kebutuhan spesifik pemrosesan data, khususnya pada penerapan *machine learning* dan peramalan performa.

## DAFTAR PUSTAKA

- [1] Y. Samadi, M. Zbakh, dan C. Tadonki, “Performance comparison between Hadoop and Spark frameworks using HiBench benchmarks”, *Concurrency and Computation: Practice and Experience*, vol. 30, no. 12, e4367, 2018. (dikunjungi pd. 09/21/2023).
- [2] D. Reinsel, J. Gantz, dan J. Rydning, “The Digitization of the World from Edge to Core”, 2018.
- [3] C. Adrian, R. Abdullah, R. Atan, dan Y. Y. Jusoh, “Expert Review on Big Data Analytics Implementation Model in Data-driven Decision-Making”, di dalam *2018 Fourth International Conference on Information Retrieval and Knowledge Management (CAMP)*, Kota Kinabalu, Malaysia: IEEE, Mar. 2018, hlmn. 1–5. (dikunjungi pd. 01/17/2024).
- [4] B. E. Syahputra dan A. Afnan, “Pendeteksian Fraud: Peran Big Data dan Audit Forensik”, *Jurnal ASET (Akuntansi Riset)*, vol. 12, no. 2, hlmn. 301–316, Des. 2020. (dikunjungi pd. 01/18/2024).
- [5] T. W. Sulaiman, R. B. Fitriansyah, A. R. Alaudin, dan M. H. Ratsanjani, “LITERATURE REVIEW: PENERAPAN BIG DATA DALAM KESEHATAN MASYARAKAT”, vol. 1, 2023.
- [6] N. Fernando, M. Mery, J. Jessica, dan J. Andry, “Utilization of Big Data In E-Commerce Business”, *Conference Series*, vol. 3, hlmn. 62–67, Nov. 2020.
- [7] R. Shrivastava dan D. S. Sisodia, “Product Recommendations Using Textual Similarity Based Learning Models”, di dalam *2019 International Conference on Computer Communication and Informatics (ICCCI)*, Jan. 2019, hlmn. 1–7. (dikunjungi pd. 05/16/2024).
- [8] *KOMPARASI KECEPATAN HADOOP MAPREDUCE DAN APACHE SPARK DALAM MENGOLAH DATA TEKS* | *Jurnal Ilmiah Matrik*, <https://journal.binadarma.ac.id/index.php/jurnalmatrik/article/view/1649>. (dikunjungi pd. 09/21/2023).
- [9] M. Saadoon, S. H. Ab. Hamid, H. Sofian, H. H. M. Altarturi, Z. H. Azizul, dan N. Nasuha, “Fault tolerance in big data storage and processing systems: A review on challenges and solutions”, *Ain Shams Engineering Journal*, vol. 13, no. 2, hlmn. 101 538, Mar. 2022. (dikunjungi pd. 01/18/2024).

- [10] R. Saputro, A. Aminuddin, dan Y. Munarko, “Perbandingan Kinerja Komputasi Hadoop dan Spark untuk Memprediksi Cuaca (Studi Kasus : Storm Event Database)”, *Jurnal Reppositor*, vol. 2, hlmn. 463, Mar. 2020.
- [11] *Hadoop Market Size, Share & Competition Analysis, 2021-2027*, <https://www.kbvresearch.com/hadoop-market/>. (dikunjungi pd. 05/15/2024).
- [12] J. Dean dan S. Ghemawat, “MapReduce: Simplified data processing on large clusters”, di dalam *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, Ser. OSDI’04, USA: USENIX Association, Des. 2004, hlmn. 10. (dikunjungi pd. 09/28/2023).
- [13] Y. Samadi, M. Zbakh, dan C. Tadonki, “Comparative study between Hadoop and Spark based on Hibench benchmarks”, di dalam *2016 2nd International Conference on Cloud Computing Technologies and Applications (CloudTech)*, Marrakech, Morocco: IEEE, Mei 2016, hlmn. 267–275. (dikunjungi pd. 09/24/2023).
- [14] H. Ahmadvand, M. Goudarzi, dan F. Foroutan, “Gapprox: Using Gallup approach for approximation in Big Data processing”, *Journal of Big Data*, vol. 6, no. 1, hlmn. 20, Feb. 2019. (dikunjungi pd. 09/29/2023).
- [15] S. Huang, J. Huang, J. Dai, T. Xie, dan B. Huang, “The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis”,
- [16] N. Ahmed, A. L. C. Barczak, T. Susnjak, dan M. A. Rashid, “A comprehensive performance analysis of Apache Hadoop and Apache Spark for large scale data sets using HiBench”, *Journal of Big Data*, vol. 7, no. 1, hlmn. 110, Des. 2020. (dikunjungi pd. 11/23/2023).
- [17] S. Gopalani dan R. Arora, “Comparing Apache Spark and Map Reduce with Performance Analysis using K-Means”, *International Journal of Computer Applications*, vol. 113, no. 1, hlmn. 8–11, Mar. 2015. (dikunjungi pd. 10/13/2023).
- [18] A. Barosen dan S. Dalin, *Analysis and Comparison of Interfacing, Data Generation and Workload Implementation in BigDataBench 4.0 and Intel Hi-Bench 7.0*. 2018. (dikunjungi pd. 12/28/2023).
- [19] A. Oussous, F.-Z. Benjelloun, A. Ait Lahcen, dan S. Belfkikh, “Big Data technologies: A survey”, *Journal of King Saud University - Computer and Information Sciences*, vol. 30, no. 4, hlmn. 431–448, Okt. 2018. (dikunjungi pd. 12/28/2023).

- [20] B. Furht dan F. Villanustre, “Introduction to Big Data”, di dalam Sept. 2016, hlmn. 3–11.
- [21] A. K. Sandhu, “Big data with cloud computing: Discussions and challenges”, *Big Data Mining and Analytics*, vol. 5, no. 1, hlmn. 32–40, Mar. 2022. (dikunjungi pd. 12/28/2023).
- [22] D. T. Vijayakumar, M. R. Vinothkanna, dan D. M. Duraipandian, “Fusion based Feature Extraction Analysis of ECG Signal Interpretation - A Systematic Approach”, *Journal of Artificial Intelligence and Capsule Networks*, vol. 3, no. 1, hlmn. 1–16, Mar. 2021. (dikunjungi pd. 05/16/2024).
- [23] S. Shaker, D. Alhajim, A. Al-khazaali, H. Hussein, dan A. Athab, “Feature Extraction based Text Classification: A review”, *Journal of Algebraic Statistics*, vol. 13, hlmn. 646–653, Mei 2022.
- [24] S. Qaiser dan R. Ali, “Text Mining: Use of TF-IDF to Examine the Relevance of Words to Documents”, *International Journal of Computer Applications*, vol. 181, no. 1, hlmn. 25–29, Juli 2018. (dikunjungi pd. 05/16/2024).
- [25] *About | DigitalOcean*, <https://www.digitalocean.com/about>. (dikunjungi pd. 05/07/2024).
- [26] C. Newham, B. Rosenblatt, dan B. Rosenblatt, *Learning the Bash Shell: Unix Shell Programming* (UNIX Shell Programming), 3. ed. Beijing Köln: O'Reilly, 2005.
- [27] K. Kalia dan N. Gupta, “Analysis of hadoop MapReduce scheduling in heterogeneous environment”, *Ain Shams Engineering Journal*, vol. 12, no. 1, hlmn. 1101–1110, Mar. 2021. (dikunjungi pd. 11/08/2023).
- [28] K. C dan A. X, “Task failure resilience technique for improving the performance of MapReduce in Hadoop”, *ETRI Journal*, vol. 42, no. 5, hlmn. 748–760, 2020. (dikunjungi pd. 11/08/2023).
- [29] H. Herodotou, *Hadoop Performance Models*, Juni 2011. arXiv: 1106.0940 [cs]. (dikunjungi pd. 11/08/2023).
- [30] M. Bakratsas, P. Basaras, D. Katsaros, dan L. Tassiulas, “Hadoop MapReduce Performance on SSDs for Analyzing Social Networks”, *Big Data Research*, vol. 11, hlmn. 1–10, Mar. 2018. (dikunjungi pd. 11/08/2023).
- [31] A. Gandomi, M. Reshadi, A. Movaghar, dan A. Khademzadeh, “HybSMRP: A hybrid scheduling algorithm in Hadoop MapReduce framework”, *Journal of Big Data*, vol. 6, no. 1, hlmn. 106, Nov. 2019. (dikunjungi pd. 11/08/2023).

- [32] *Apache Hadoop*, <https://hadoop.apache.org/>. (dikunjungi pd. 11/08/2023).
- [33] S. Maneas dan B. Schroeder, “The Evolution of the Hadoop Distributed File System”, di dalam *2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, Mei 2018, hlmn. 67–74. (dikunjungi pd. 11/08/2023).
- [34] C. Dabas, P. Kaur, N. Gulati, dan M. Tilak, “Analysis of Comments on YouTube Videos using Hadoop”, di dalam *2019 Fifth International Conference on Image Information Processing (ICIIP)*, Nov. 2019, hlmn. 353–358. (dikunjungi pd. 11/08/2023).
- [35] T. John dan P. Misra, *Data Lake for Enterprises: Leveraging Lambda Architecture for Building Enterprise Data Lake*. Birmingham, UK, Mumbai: Packt Publishing, 2017.
- [36] S. Khatai, S. S. Rautaray, S. Sahoo, dan M. Pandey, “An Implementation of Text Mining Decision Feedback Model Using Hadoop MapReduce”, di dalam *Trends of Data Science and Applications: Theory and Practices*, Ser. Studies in Computational Intelligence, S. S. Rautaray, P. Pemmaraju, dan H. Mohanty, timed., Singapore: Springer, 2021, hlmn. 273–306. (dikunjungi pd. 11/09/2023).
- [37] K. Abhishek, Department of CSE, NIT Patna, Ashok Rajpath, Mahendru, Patna - 800005, Bihar, India, M. Kumar Verma, dkk., “Integrated Hadoop Cloud Framework (IHCF)”, *Indian Journal of Science and Technology*, vol. 10, no. 10, hlmn. 1–8, Feb. 2017. (dikunjungi pd. 11/10/2023).
- [38] *Apache Hadoop 3.3.6 – HDFS Architecture*, <https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>. (dikunjungi pd. 11/10/2023).
- [39] H. T. Almansouri dan Y. Masmoudi, “Hadoop Distributed File System for Big data analysis”, di dalam *2019 4th World Conference on Complex Systems (WCCS)*, Ouarzazate, Morocco: IEEE, Apr. 2019, hlmn. 1–5. (dikunjungi pd. 11/08/2023).
- [40] *Apache Hadoop 3.3.6 – Apache Hadoop YARN*, <https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html>. (dikunjungi pd. 12/28/2023).
- [41] *Apache Spark™ - Unified Engine for large-scale data analytics*, <https://spark.apache.org/>. (dikunjungi pd. 11/10/2023).

- [42] *Apache Spark - Introduction*, [https://www.tutorialspoint.com/apache\\_spark.htm](https://www.tutorialspoint.com/apache_spark.htm). (dikunjungi pd. 12/30/2023).
- [43] *Apache Spark - RDD*, [https://www.tutorialspoint.com/apache\\_spark/apache\\_spark\\_rdd.htm](https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm). (dikunjungi pd. 01/15/2024).
- [44] *Intel-bigdata/HiBench*, Intel-bigdata, Des. 2023. (dikunjungi pd. 12/30/2023).
- [45] *MapReduce - Distributed Computing in Java 9 [Book]*, <https://www.oreilly.com/library/view/distributed-computing-in/9781787126992/5fef6ce5-20d7-4d7c-93eb-7e669d48c2b4.xhtml>. (dikunjungi pd. 11/08/2023).

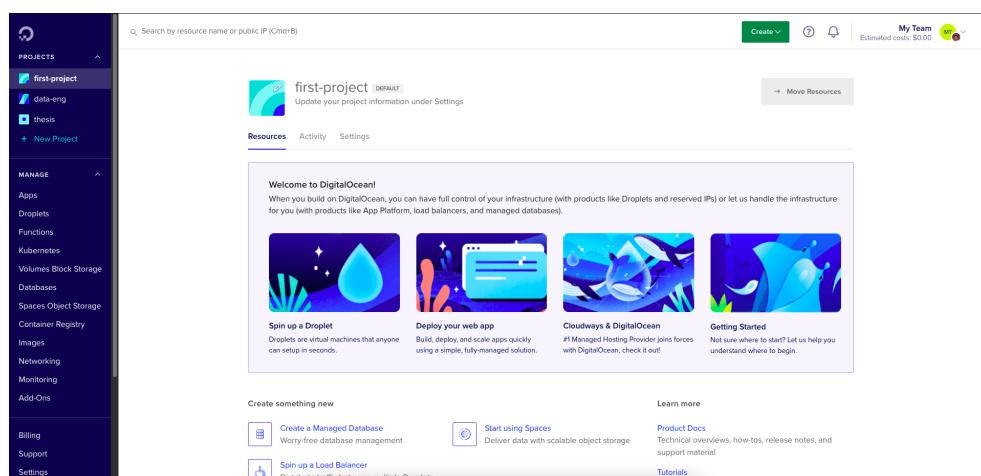
## LAMPIRAN

# LAMPIRAN A

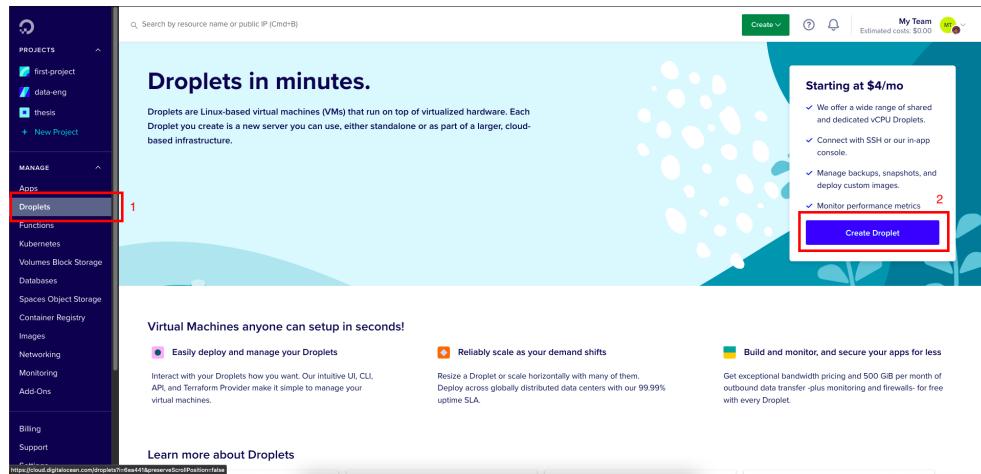
## Pembuatan *Virtual Machine* (VM) pada DigitalOcean

Langkah-langkah pembuatan VM pada DigitalOcean dijelaskan seperti berikut,

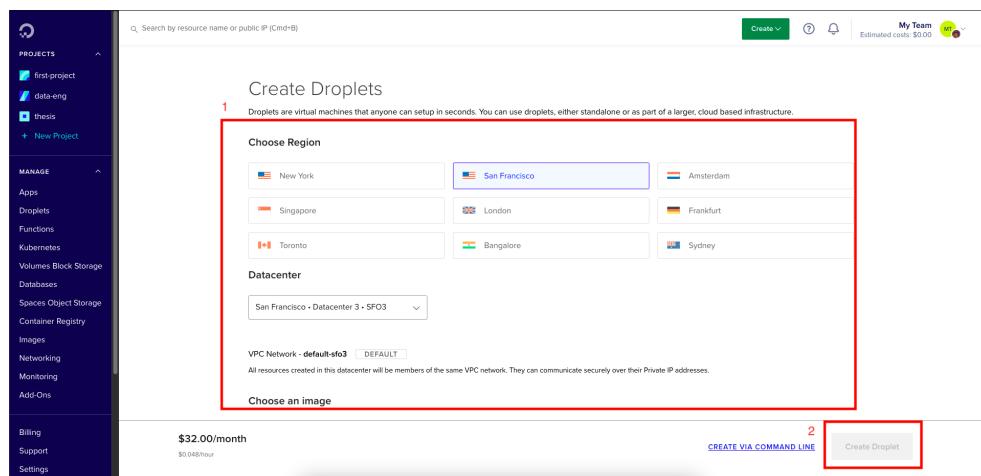
1. Buatlah akun DigitalOcean terlebih dahulu. Jika belum memiliki akun DigitalOcean, disarankan untuk mendaftar melalui *GitHub Student Developer Pack* sehingga nantinya akan diberikan kredit \$200 secara gratis. Jika sudah memiliki akun DigitalOcean, silakan melakukan *login*.
2. Halaman dasbor DigitalOcean akan ditampilkan setelahnya.



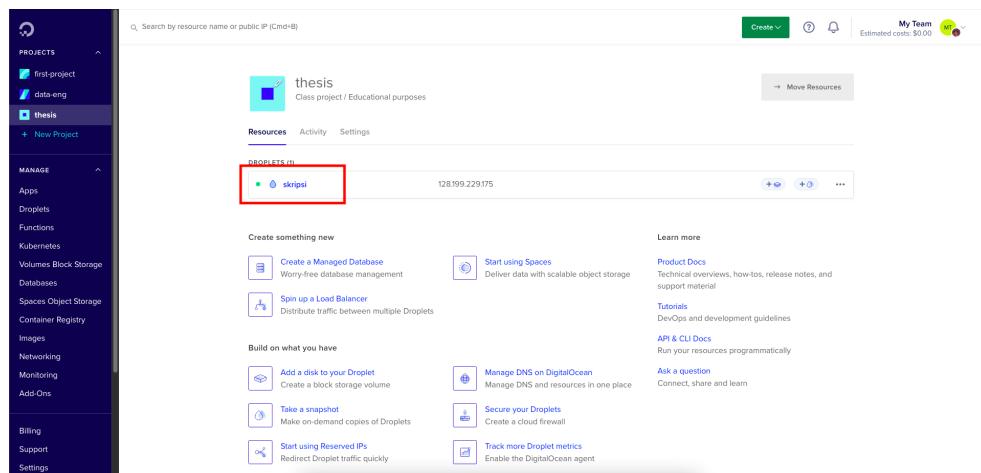
3. Perhatikan menu di sebelah kiri pada laman dasbor DigitalOcean. Tekan *Droplets* untuk masuk ke laman pembuatan VM. Selanjutnya, tekan *Create Droplets* berwarna biru untuk melakukan konfigurasi VM yang akan dibuat nantinya.



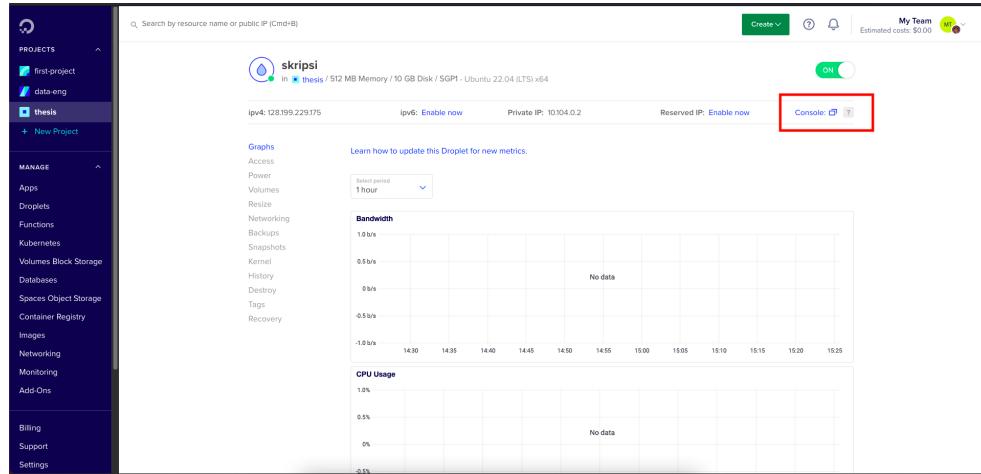
4. Pada laman pembuatan Droplets, lakukan konfigurasi sesuai dengan Tabel 3.1. Jika telah selesai melakukan konfigurasi, tekan *Create Droplets*.



5. Jika pembuatan Droplets berhasil, laman dasbor *Projects* DigitalOcean akan terlihat. Pada bagian Resources akan terlihat Droplets yang baru saja kita buat. Selanjutnya, tekan nama Droplets yang baru saja dibuat.



- Selanjutnya, laman konfigurasi Droplets akan terlihat. Jika diperlukan konfigurasi lanjutan dapat diatur melalui laman ini. Pada tahap ini hanya akan fokus pada konfigurasi perangkat lunak tanpa konfigurasi perangkat keras lebih jauh. Untuk masuk ke VM yang sudah dibuat, tekan Console. Tab baru akan dibuka.



- Akhirnya, lakukan konfigurasi perangkat lunak pada bagian ini.

```

skripsi - DigitalOcean Droplet Web Console
https://cloud.digitalocean.com/droplets/39235116/terminal/ui/
Welcome to Ubuntu 22.04.2 LTS (GNU/Linux 5.15.0-67-generic x86_64)

 * Documentation: https://help.ubuntu.com
 * Management: https://landscape.canonical.com
 * Support: https://ubuntu.com/advantage

System information disabled due to load higher than 1.0
Expanded Security Maintenance for Applications is not enabled.

17 updates can be applied immediately.
13 of these updates are standard security updates.
To see these additional updates run: apt list --upgradable

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status

The list of available updates is more than a week old.
To check for new updates run: sudo apt update

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

root@skripsi:~# 

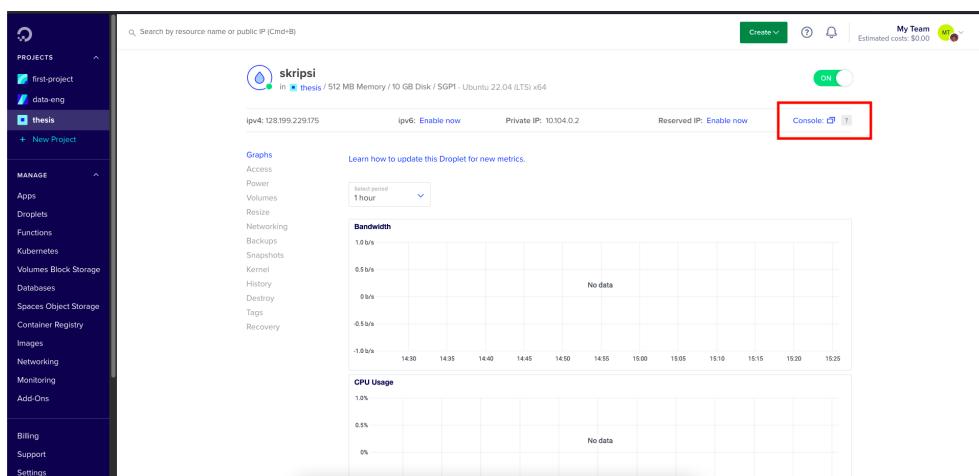
```

## LAMPIRAN B

### Instalasi dan Konfigurasi Perangkat Lunak Prasyarat

Pemasangan dan konfigurasi perangkat lunak adalah hal yang krusial. Sebelum dilakukan pemasangan perangkat lunak penyimpanan dan pemrosesan *big data*, tentunya perlu disiapkan perangkat lunak prasyarat. Perangkat lunak prasyarat yang dibutuhkan meliputi Git, Java dan Maven, Python, serta Scala. Langkah-langkah pemasangan dan konfigurasi perangkat lunak akan dijelaskan sebagai berikut,

1. Pastikan Droplets pada DigitalOcean sudah dibuat. Masuk ke *Virtual Machine* (VM) yang sebelumnya sudah dibuat melalui *Console* yang berada pada laman konfigurasi Droplets DigitalOcean.



2. Jika Droplets baru saja dibuat, perlu dilakukan pembaruan *index* pada *package management*. *Package management* adalah sistem atau sekumpulan alat yang digunakan untuk mengotomatiskan penginstalan, peningkatan, konfigurasi, dan penggunaan perangkat lunak. Pembaruan *package management* dapat dilakukan dengan `sudo apt update`.
3. Membuat Pengguna Baru
  - (a) Pertama, buatlah grup baru yang bernama *hadoop* dengan perintah `sudo addgroup hadoop`.
  - (b) Kemudian, tambahkan pengguna baru *hdfsuser* dalam grup *hadoop* yang sama dengan perintah `sudo adduser --ingroup hadoop hdfsuser`.
  - (c) Berikan *hdfsuser* izin *root* yang diperlukan untuk pemasangan file. Hak istimewa pengguna *root* dapat diberikan dengan memperbarui file *sudoers*. Buka file *sudoers* dengan menjalankan perintah `sudo visudo`. Tambahkan baris berikut, yaitu `hdfsuser ALL=(ALL:ALL) ALL`.

- (d) Sekarang, simpan perubahan dan tutup editor.
- (e) Selanjutnya, mari beralih ke pengguna baru yang telah dibuat untuk instalasi lebih lanjut menggunakan perintah `su - hdfsuser`.
4. Pengaturan *SSH keys* untuk Hadoop
- (a) Hadoop menggunakan *Secure Shell* (SSH) untuk menjalankan proses antara *master nodes* dan *slave nodes*. Penggunaan SSH akan memberikan banyak keuntungan, salah satunya adalah kecepatan. Jika sebuah klaster aktif dan berjalan, komunikasi antar *nodes* akan berjalan terlalu sering. Begitu pula dengan *job tracker* yang harus sering mengirimkan informasi *task to task* dengan cepat. Lakukan pemasangan ssh dan sshd dengan cara `sudo apt-get install ssh` dan `sudo apt-get install sshd` pada terminal.
- (b) Selanjutnya, lakukan pembuatan *SSH keys* dengan cara `ssh-keygen -t rsa`. Jika pembuatan *SSH keys* sudah dilakukan, jalankan perintah `cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys`.
- (c) Ubah perizinan berkas dengan perintah `chmod og-wx ~/.ssh/authorized_keys`.
- (d) Terakhir, untuk memverifikasi koneksi aman sudah terjadi, lakukan `ssh localhost`.
5. Instalasi Git
- (a) Git dapat dipasang menggunakan perintah `sudo apt install git`. Pengguna akan diminta konfirmasi untuk menginstall. Ketik `y` kemudian tekan enter.
- (b) Untuk mengecek versi Git, dapat menggunakan perintah `git --version`.
6. Instalasi Python
- (a) Python dapat dipasang menggunakan perintah `sudo apt-get install python2 && sudo apt install python3.7`. Pengguna akan diminta konfirmasi untuk menginstall. Ketik `y` kemudian tekan enter.
- (b) Untuk mengecek versi Python, dapat menggunakan perintah `python --version`.
7. Instalasi Java 8 dan Maven
- (a) Java 8 dapat dipasang menggunakan perintah `sudo apt install openjdk-8-jre-headless openjdk-8-jdk`. Pengguna akan diminta konfirmasi untuk menginstall. Ketik `y`

kemudian tekan enter.

- (b) Versi dari Java dapat dilihat menggunakan perintah `java -version`.
- (c) Selanjutnya, instalasi Maven dapat dilakukan menggunakan perintah `sudo apt-get -y install maven`.
- (d) Informasi dari Maven beserta Java yang digunakan dapat dilihat menggunakan perintah `mvn -version`.

8. Instalasi Scala

- (a) Scala yang akan dipasang adalah versi 2.12. Jika menggunakan manajer paket, versi yang akan dipasang adalah versi terbaru. Untuk mengunduh versi spesifik dari Scala, dapat menggunakan perintah `sudo wget https://downloads.lightbend.com/scala/2.12.0/scala-2.12.0.deb`.
- (b) Scala dapat dipasang menggunakan perintah `sudo dpkg -i scala-2.12.0.deb`.
- (c) Versi Scala dapat dilihat melalui perintah `scala -version`.

## LAMPIRAN C

### Instalasi dan Konfigurasi Hadoop

Langkah-langkah pemasangan dan konfigurasi Hadoop akan dijelaskan sebagai berikut,

1. Unduh Hadoop
  - (a) Pastikan perangkat lunak prasyarat sudah berhasil dipasang dan dilakukan konfigurasi. Sebelum dilakukan pemasangan Hadoop, diperlukan untuk mengunduh berkas Hadoop terlebih dahulu dengan perintah `cd /usr/local`, dilanjutkan dengan `sudo wget https://archive.apache.org/dist/hadoop/common/hadoop-2.4.0/hadoop-2.4.0.tar.gz`.
  - (b) Ekstrak berkas Hadoop yang sudah diunduh tadi dengan perintah `sudo tar xvzf hadoop-2.4.0.tar.gz`. Hasil ekstrak berkas Hadoop akan disimpan pada direktori yang sama.
  - (c) Selanjutnya, untuk memudahkan kedepannya, ganti nama folder Hadoop dengan perintah `sudo mv hadoop-2.4.0 hadoop`.
2. Mengubah Kepemilikan Berkas Hadoop
  - (a) Setelah berkas Hadoop sudah berhasil terunduh, selanjutnya ubah kepemilikan berkas Hadoop ke `hdfsuser` yang sebelumnya sudah kita buat dengan perintah `sudo chown -R hdfsuser:hadoop /usr/local/hadoop`.
  - (b) Tambahkan kekuasaan untuk membaca, menulis, dan mengeksekusi pada foler Hadoop dengan perintah `sudo chmod -R 777 /usr/local/hadoop`.
3. Mematikan *IPv6 Networks*
  - (a) Saat ini Hadoop belum mendukung penggunaan *IPv6 Networks*. Hadoop hanya dibangun dan diuji coba pada *IPv4 Networks*. Untuk mematikan IPv6, dapat dimulai dengan menjalankan perintah `cat /proc/sys/net/ipv6/conf/all/disable_ipv6`.
  - (b) Jika hasil yang diberikan bukan angka 1, maka beberapa langkah tambahan harus dijalankan. Jalankan perintah `sudo nano /etc/sysctl.conf`, kemudian tambahkan beberapa baris potongan kode berikut pada akhir berkas,

---

```
1      # Disable ipv6
2      net.ipv6.conf.all.disable_ipv6=1
3      net.ipv6.conf.default_ipv6=1
```

```
4     net.ipv6.conf.lo.disable_ipv6=1
```

---

- (c) Simpan berkas. Kemudian jalankan perintah `sudo sysctl -p` untuk mengaktifkan perubahan.

#### 4. Menambahkan Hadoop pada *Environments Variables*

- (a) Hadoop perlu ditambahkan pada *Environments Variables* untuk memudahkan dalam melakukan eksekusi. Untuk menambahkannya, jalankan perintah `sudo nano ~/.bashrc`.
- (b) Tambahkan beberapa baris kode berikut pada akhir berkas *bashrc*.
- 

```
1      # HADOOP ENVIRONMENT
2      export HADOOP_HOME=/usr/local/hadoop
3      export HADOOP_CONF_DIR=/usr/local/hadoop/etc/hadoop
4      export HADOOP_MAPRED_HOME=/usr/local/hadoop
5      export HADOOP_COMMON_HOME=/usr/local/hadoop
6      export HADOOP_HDFS_HOME=/usr/local/hadoop
7      export YARN_HOME=/usr/local/hadoop
8      export PATH=$PATH:/usr/local/hadoop/bin
9      export PATH=$PATH:/usr/local/hadoop/sbin
10
11     # HADOOP NATIVE PATH
12     export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/←
13         lib/native
14     export HADOOP_OPTS=-Djava.library.path=←
15         $HADOOP_PREFIX/lib
```

---

- (c) Untuk mendapatkan perubahan dapat dilakukan dengan perintah `source ~/.bashrc`.

#### 5. Konfigurasi Hadoop

- (a) Hadoop menggunakan berkas .xml untuk melakukan konfigurasi pada semua prosesnya. Biasanya, letak direktori untuk melakukan konfigurasi terletak pada `$HADOOP_HOME/etc/hadoop`. Oleh karena itu, jalankan perintah `cd /usr/local/hadoop/etc/hadoop/`.
- (b) Konfigurasi berkas *hadoop-env.sh* dapat dilakukan dengan perintah `sudo nano hadoop-env.sh`, dilanjutkan dengan menambahkan beberapa baris kode seperti di bawah ini,
- 

```
1      export HADOOP_OPTS=-Djava.net.preferIPv4Stack=true
2      export JAVA_HOME=/usr
3      export HADOOP_HOME_WARN_SUPPRESS="TRUE"
4      export HADOOP_ROOT_LOGGER="WARN,DRFA"
5      export HDFS_NAMENODE_USER="hdfsuser"
6      export HDFS_DATANODE_USER="hdfsuser"
7      export HDFS_SECONDARYNAMENODE_USER="hdfsuser"
```

---

```
8     export YARN_RESOURCEMANAGER_USER="hdfsuser"
9     export YARN_NODEMANAGER_USER="hdfsuser"
```

---

- (c) Konfigurasi berkas *yarn-site.xml* dapat dilakukan dengan perintah **sudo nano yarn-site.xml**, dilanjutkan dengan menambahkan beberapa baris kode seperti di bawah ini,
- 

```
1 <property>
2   <name>yarn.nodemanager.aux-services</name>
3   <value>mapreduce_shuffle</value>
4 </property>
5 <property>
6   <name>yarn.nodemanager.aux-services.mapreduce.<-
        shuffle.class</name>
7   <value>org.apache.hadoop.mapred.ShuffleHandler</<-
        value>
8 </property>
```

---

- (d) Konfigurasi berkas *hdfs-site.xml* dapat dilakukan dengan perintah **sudo nano hdfs-site.xml**, dilanjutkan dengan menambahkan beberapa baris kode seperti di bawah ini,
- 

```
1 <property>
2   <name>dfs.replication</name>
3   <value>1</value>
4 </property>
5 <property>
6   <name>dfs.namenode.name.dir</name>
7   <value>/usr/local/hadoop/yarn_data/hdfs/namenode</<-
        value>
8 </property>
9 <property>
10  <name>dfs.datanode.data.dir</name>
11  <value>/usr/local/hadoop/yarn_data/hdfs/datanode</<-
        value>
12 </property>
13 <property>
14  <name>dfs.namenode.http-address</name>
15  <value>localhost:50070</value>
16 </property>
```

---

- (e) Konfigurasi berkas *core-site.xml* dapat dilakukan dengan perintah **sudo nano core-site.xml**, dilanjutkan dengan menambahkan beberapa baris kode seperti di bawah ini,
- 

```
1 <property>
```

```
2      <name>hadoop.tmp.dir</name>
3      <value>/bigdata/hadoop/tmp</value>
4      </property>
5      <property>
6          <name>fs.default.name</name>
7          <value>hdfs://localhost:9000</value>
8      </property>
```

---

- (f) Konfigurasi berkas *mapred-site.xml* dapat dilakukan dengan perintah `sudo nano mapred-site.xml`, dilanjutkan dengan menambahkan beberapa baris kode seperti di bawah ini,

```
1      <property>
2          <name>mapred.framework.name</name>
3          <value>yarn</value>
4      </property>
5      <property>
6          <name>mapreduce.jobhistory.address</name>
7          <value>localhost:10020</value>
8      </property>
```

---

## 6. Membuat Direktori Hadoop untuk Menyimpan Data

- (a) Sesuai dengan apa yang ditulis pada *core-site.xml*, langkah pertama yang harus dilakukan adalah membuat direktori sementara untuk dfs menyimpan berkas dengan menjalankan perintah di bawah. Jalankan perintah berikut baris per baris.

```
1      sudo mkdir -p /bigdata/hadoop/tmp
2      sudo chown -R hdfsuser:hadoop /bigdata/hadoop/tmp
3      sudo chmod -R 777 /bigdata/hadoop/tmp
```

---

- (b) Selanjutnya, jalankan perintah berikut untuk membuat direktori untuk menyimpan berkas data sekaligus mengganti kepemilikan berkas. Jalankan perintah berikut baris per baris.

```
1      sudo mkdir -p /usr/local/hadoop/yarn_data/hdfs/←
        namenode
2      sudo mkdir -p /usr/local/hadoop/yarn_data/hdfs/←
        datanode
3      sudo chmod -R 777 /usr/local/hadoop/yarn_data/hdfs/←
        namenode
4      sudo chmod -R 777 /usr/local/hadoop/yarn_data/hdfs/←
        datanode
5      sudo chown -R hdfsuser:hadoop /usr/local/hadoop/←
        yarn_data/hdfs/namenode
```

---

```
6      sudo chown -R hdfsuser:hadoop /usr/local/hadoop/←  
          yarn_data/hdfs/datanode
```

---

- (c) Konfigurasi untuk Hadoop sudah selesai dan dapat dilanjutkan untuk menjalankan *Resource Manager* dan *Node Manager*
7. Menjalankan Hadoop
- (a) Sebelum menjalankan *Hadoop Core Services*, klaster harus dibersihkan dengan cara melakukan *format* pada *namenode*. Jalankan perintah `hdfs namenode -format`.
  - (b) Untuk menjalankan layanan Hadoop, dapat dilakukan dengan perintah `start-all.sh`.
  - (c) Perintah `jps` dapat dilakukan untuk mengecek apakah layanan Hadoop sudah berjalan.
  - (d) Untuk memberhentikan layanan Hadoop, dapat dilakukan dengan perintah `stop-all.sh` pada terminal.

## LAMPIRAN D

### Instalasi dan Konfigurasi Spark

Langkah-langkah pemasangan dan konfigurasi Spark akan dijelaskan sebagai berikut,

1. Unduh Berkas Spark
  - (a) Pastikan perangkat lunak prasyarat sudah berhasil dipasang dan dilakukan konfigurasi. Sebelum dilakukan pemasangan Spark, diperlukan untuk mengunduh berkas Spark terlebih dahulu dengan perintah `cd /usr/local`, dilanjutkan dengan `sudo wget https://archive.apache.org/dist/spark/spark-2.1.3/spark-2.1.3-bin-hadoop2.4.tgz`.
  - (b) Ekstrak berkas Spark yang sudah diunduh tadi dengan perintah `sudo tar xvf spark-2.1.3-bin-hadoop2.4.tgz`. Hasil ekstrak berkas Spark akan disimpan pada direktori yang sama.
  - (c) Selanjutnya, untuk memudahkan kedepannya, ganti nama folder Spark dengan perintah `sudo mv spark-2.1.3-bin-hadoop2.4 spark`.
2. Menambahkan Spark pada *Environments Variables*
  - (a) Spark perlu ditambahkan pada *Environments Variables* untuk memudahkan dalam melakukan eksekusi. Untuk menambahkannya, jalankan perintah `sudo nano ~/.bashrc`.
  - (b) Tambahkan beberapa baris kode berikut pada akhir berkas *bashrc*.

---

```
1 # SPARK ENVIRONMENT
2 export PATH=$PATH:/usr/local/spark/bin
3 export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
4 export SPARK_HOME=$PATH:/usr/local/spark/bin
```

---
  - (c) Untuk mendapatkan perubahan dapat dilakukan dengan perintah `source ~/.bashrc`.
3. Menjalankan *Spark Shell*
  - (a) Pastikan bahwa Spark sudah ditambahkan pada *environments variables* dengan perintah `spark-submit --version`.
  - (b) Jalankan layanan Hadoop dengan perintah `start-all.sh`.
  - (c) Jalankan `spark-shell` dengan YARN menggunakan perintah `spark-shell --master yarn`.

## LAMPIRAN E

### Instalasi dan Konfigurasi HiBench

Langkah-langkah pemasangan dan konfigurasi HiBench akan dijelaskan sebagai berikut,

1. Unduh berkas HiBench
  - (a) Pastikan perangkat lunak prasyarat dan perangkat lunak *Big Data* sebelumnya sudah berhasil dipasang dan dilakukan konfigurasi. Tahap selanjutnya adalah mengunduh berkas HiBench dengan perintah `git clone https://github.com/Intel-bigdata/HiBench.git`. Pastikan berada pada folder `/home/hdfsuser`.
  - (b) Selanjutnya, berikan perizinan ke folder HiBench dengan cara `sudo chmod 755 HiBench`.
2. Membangun *framework benchmark*
  - (a) Sebelum HiBench dapat digunakan, diperlukan pembangunan beberapa modul yang dibutuhkan, misalnya modul *data generation*, modul *hadoopbench*, dan modul *sparkbench*. Langkah awal yang diperlukan adalah masuk ke folder HiBench dengan perintah `cd HiBench`.
  - (b) Selanjutnya, pastikan versi Hadoop, Spark, dan Scala sudah sesuai. Jalankan perintah `mvn -Phadoopbench -Psparkbench -Dhadoop=2.4 -Dspark=2.1 -Dmodules -Pmicro clean package`. Perintah ini akan membangun modul yang dibutuhkan menggunakan Maven. Tidak semua modul akan dipasang. Modul yang akan dipasang salah satunya adalah modul untuk *micro benchmark*.
  - (c) Jika ingin pembangunan modul yang lebih luas cakupannya dapat menggunakan perintah `mvn -Phadoopbench -Psparkbench -Dspark=2.1 -Dscala=2.11 clean package`.
3. Jalankan perintah `sudo apt install bc` supaya berkas Hibench *Report* dapat muncul.
4. Lakukan konfigurasi pada berkas *hibench.conf*. Buka berkas tersebut dengan perintah `sudo nano conf/hibench.conf`.
5. Lakukan beberapa perubahan pada baris sesuai dengan contoh di bawah ini.

---

<code>1 hibench.masters.hostnames</code>	<code>hdfs://localhost:9000</code>
<code>2 hibench.slaves.hostnames</code>	<code>localhost</code>

---

## 6. Menjalankan *Benchmark* Hadoop

- (a) Lakukan konfigurasi pada berkas *hadoop.conf*. Sebelum itu, salin *template* konfigurasinya dengan perintah `cp conf/hadoop.conf.template conf/hadoop.conf`.
- (b) Buka berkas *hadoop.conf* dengan perintah `sudo nano conf/hadoop.conf`. Selanjutnya, lakukan beberapa perubahan seperti pada contoh di bawah.
- 

```
1   # Hadoop home
2   hibench.hadoop.home      /usr/local/hadoop
3
4   # The path of hadoop executable
5   hibench.hadoop.executable ${hibench.hadoop.home}-
6   /bin/hadoop
7
8   # Hadoop configraution directory
9   hibench.hadoop.configure.dir ${hibench.hadoop.home}-
10  /etc/hadoop
11
12
13  # The root HDFS path to store HiBench data
14  hibench.hdfs.master      hdfs://localhost:9000
15
16
17  # Hadoop release provider. Supported value: apache
18  hibench.hadoop.release    apache
```

---

- (c) Simpan perubahan yang sudah dilakukan.
- (d) Untuk menjalankan beban kerja yang sudah dirancang sebelumnya, perintah yang digunakan sebagai berikut. Jalankan baris per baris.
- 

```
1   bin/workloads/micro/<nama-beban-kerja>/prepare/-
2   prepare.sh
3   bin/workloads/micro/<nama-beban-kerja>/hadoop/run-
4   .sh
```

---

- (e) Untuk melihat hasil dari *benchmark* dapat mengakses berkas pada `<HiBench_Root>/report/hibench.report`

## 7. Menjalankan *Benchmark* Spark

- (a) Lakukan konfigurasi pada berkas *spark.conf*. Sebelum itu, salin *template* konfigurasinya dengan perintah `cp conf/spark.conf.template conf/spark.conf`.
- (b) Buka berkas *spark.conf* dengan perintah `sudo nano conf/spark.conf`. Selanjutnya, lakukan beberapa perubahan seperti pada contoh di bawah.

---

```
1 # Spark home
2 hibench.spark.home      /usr/local/spark
3
4 # Spark master
5 #   standalone mode: spark://xxx:7077
6 #   YARN mode: yarn-client
7 hibench.spark.master    yarn-client
```

---

- (c) Simpan perubahan yang sudah dilakukan.
- (d) Untuk menjalankan beban kerja yang sudah dirancang sebelumnya, perintah yang digunakan sebagai berikut. Jalankan baris per baris.

---

```
1 bin/workloads/micro/<nama-beban-kerja>/prepare/←
  prepare.sh
2 bin/workloads/micro/<nama-beban-kerja>/spark/run.←
  sh
```

---

- (e) Untuk melihat hasil dari *benchmark* dapat mengakses berkas pada <HiBench\_Root>/report/hibench.report

# LAMPIRAN F

## Skrip Otomatisasi Eksperimen

---

```
1 #!/bin/bash
2
3 # Ubah direktori kerja ke direktori HiBench
4 cd /home/hadoop/HiBench-HiBench-7.0
5
6 # Daftar workload
7 workloads=("wordcount" "sort")
8
9 # Daftar skala
10 scales=(
11     "seratuskb"
12     "limaratuskb"
13     "satumb"
14     "limamb"
15     "sepuluhmb"
16     "limapuluuhmb"
17     "seratusmb"
18     "halfgig"
19     "onegig"
20     "fivegig"
21     "tengig"
22     "fiveteengig"
23 )
24
25 # Jumlah pengulangan
26 repetitions=5
27
28 # Looping untuk setiap workload
29 for workload in "${workloads[@]}"; do
30     echo "Menjalankan workload: $workload"
31
32     # Looping untuk setiap skala
33     for scale in "${scales[@]}"; do
34         echo " Skala: $scale"
35
36         # Mengubah konfigurasi HiBench
37         sed -i "s/^hibench.scale.profile.*$/hibench.scale.profile<--$scale/" conf/hibench.conf
38
39         # Tahap persiapan Hadoop
40         while true; do
```

```

41      bin/workloads/micro/$workload/prepare/prepare.sh
42      if [[ $? -eq 0 ]]; then
43          break
44      fi
45      echo "    Tahap persiapan Hadoop gagal. Mencoba lagi←
46          ..."
47      done
48
49      # Looping untuk setiap pengulangan Hadoop
50      for ((i = 1; i <= repetitions; i++)); do
51          echo "    Percobaan Hadoop $i"
52
53          # Mulai dool di latar belakang
54          nohup /home/hadoop/bin/dool --all --io --output "←
55              $workload-$scale-$i-hadoop.csv" --bytes > /dev/←
56              null 2>&1 &
57          dool_pid=$! # Menyimpan PID proses dool
58
59          # Menjalankan benchmark Hadoop
60          bin/workloads/micro/$workload/hadoop/run.sh
61
62          # Menghentikan dool setelah benchmark selesai
63          kill $dool_pid
64          wait $dool_pid 2>/dev/null
65      done
66
67      # Tahap persiapan Spark
68      while true; do
69          bin/workloads/micro/$workload/prepare/prepare.sh
70          if [[ $? -eq 0 ]]; then
71              break
72          fi
73          echo "    Tahap persiapan Spark gagal. Mencoba lagi←
74              ..."
75      done
76
77      # Looping untuk setiap pengulangan Spark
78      for ((i = 1; i <= repetitions; i++)); do
79          echo "    Percobaan Spark $i"

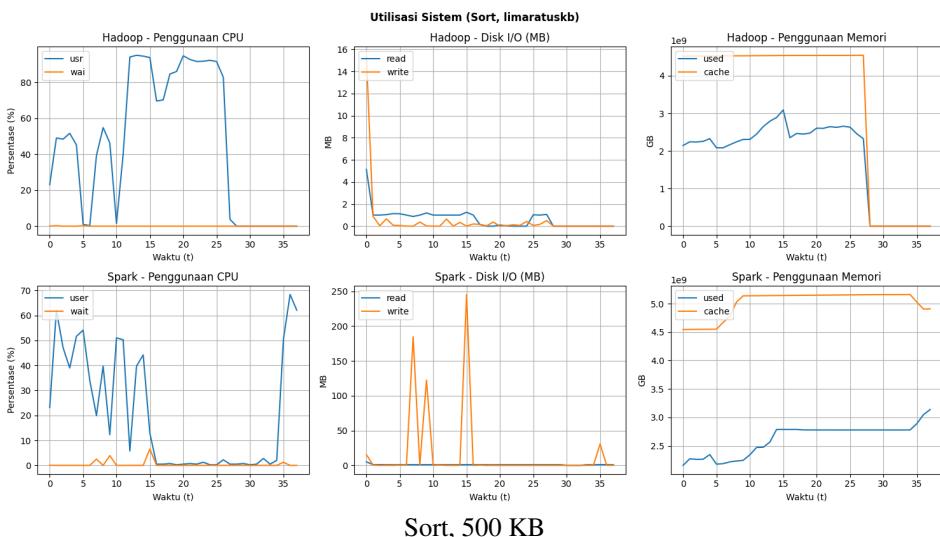
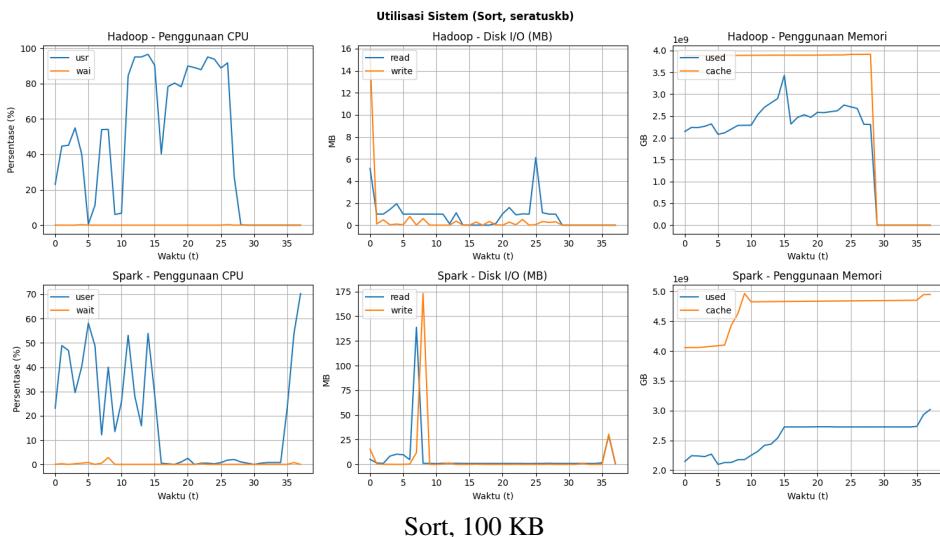
```

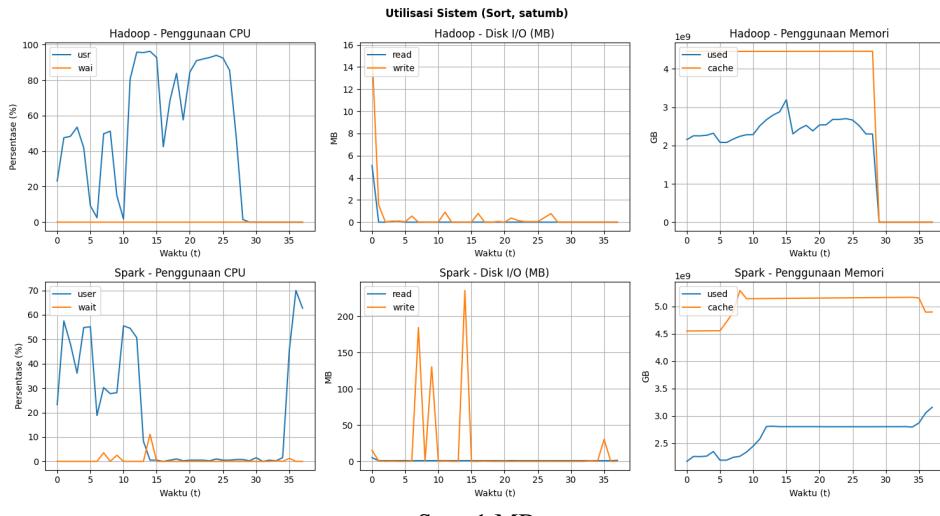
```
80
81      # Menjalankan benchmark Spark
82      bin/workloads/micro/$workload/spark/run.sh
83
84      # Menghentikan dool setelah benchmark selesai
85      kill $dool_pid
86      wait $dool_pid 2>/dev/null
87      done
88
89      # Tunggu 15 detik sebelum beralih ke skala berikutnya
90      sleep 15
91      done
92 done
93
94 echo "Selesai."
```

---

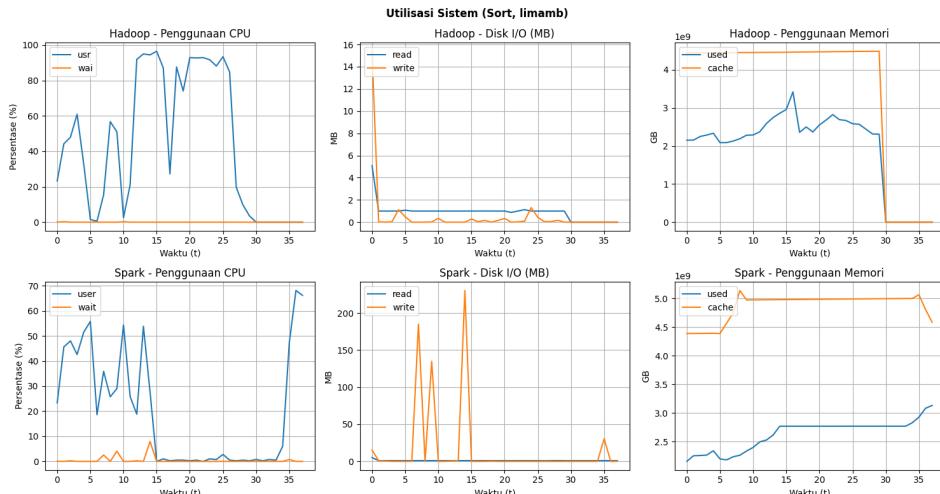
## LAMPIRAN G

### Visualisasi Utilisasi Sistem Sesuai Input Data (Sort)

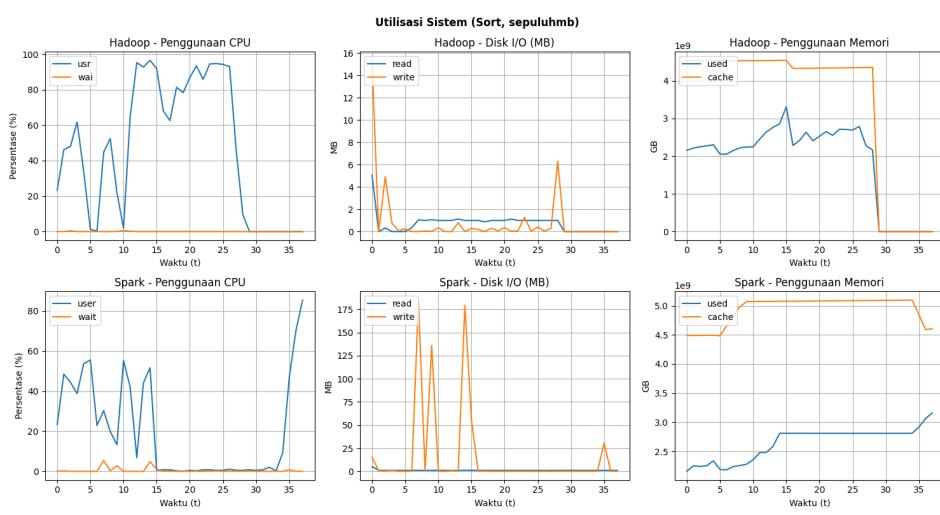




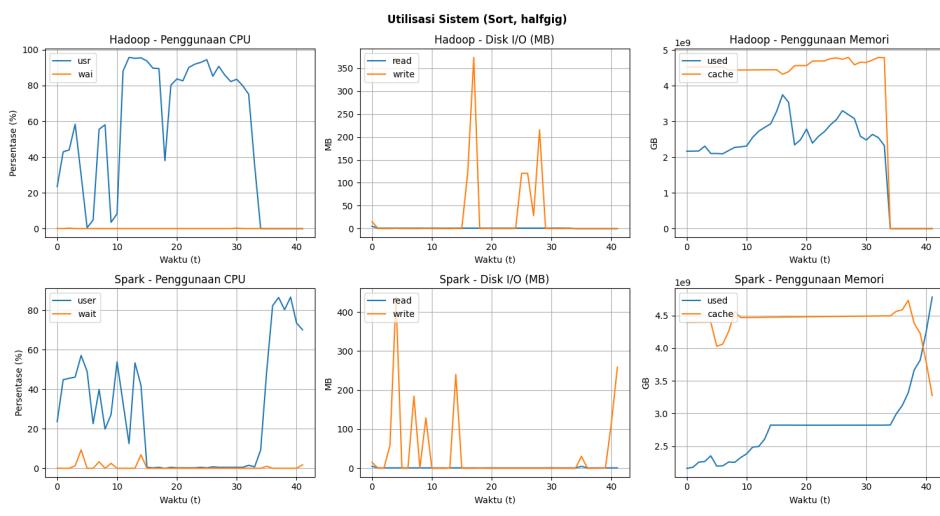
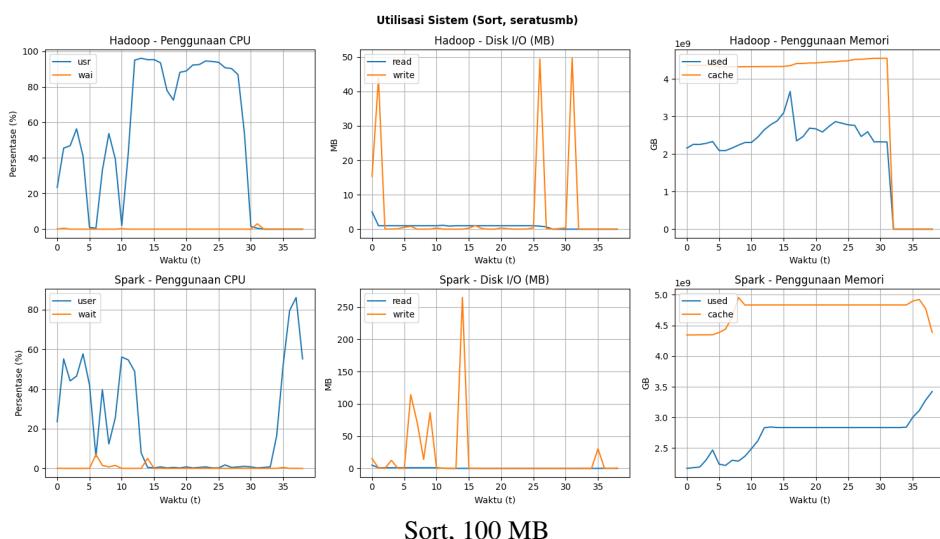
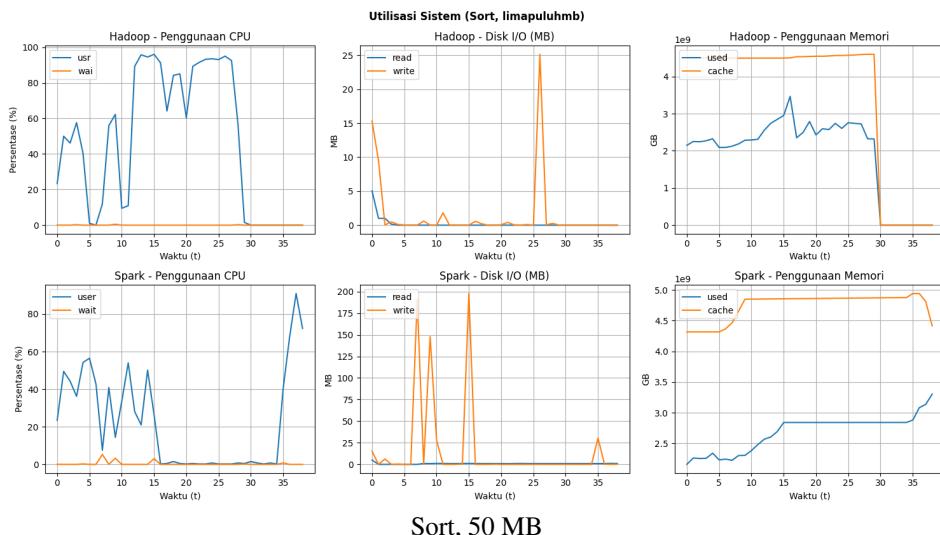
Sort, 1 MB

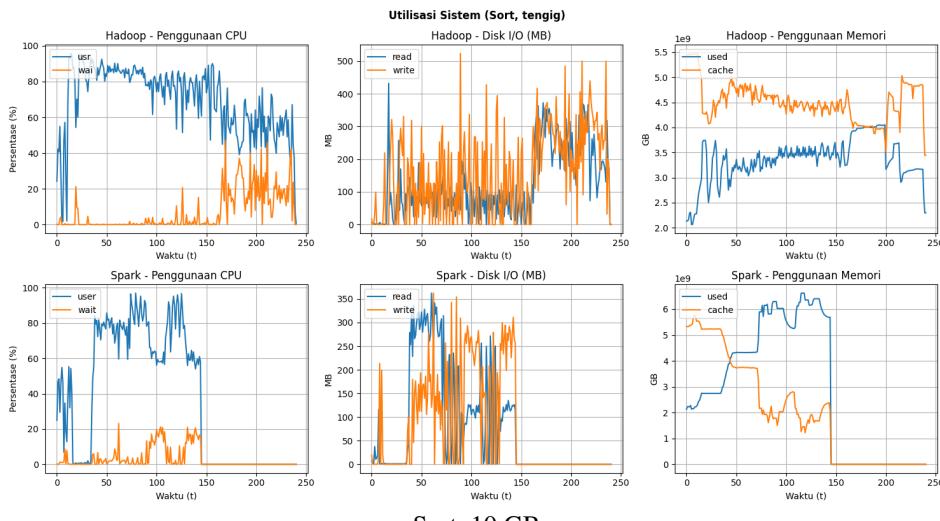
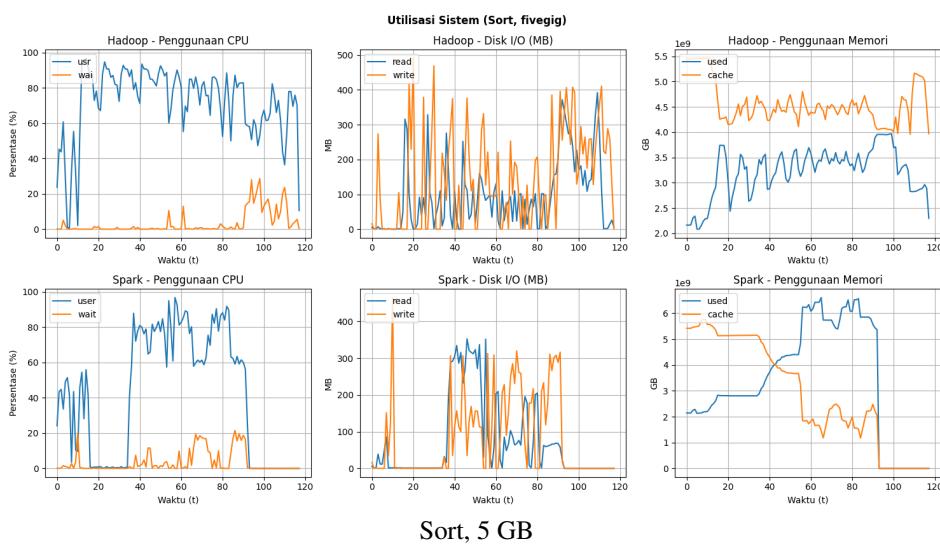
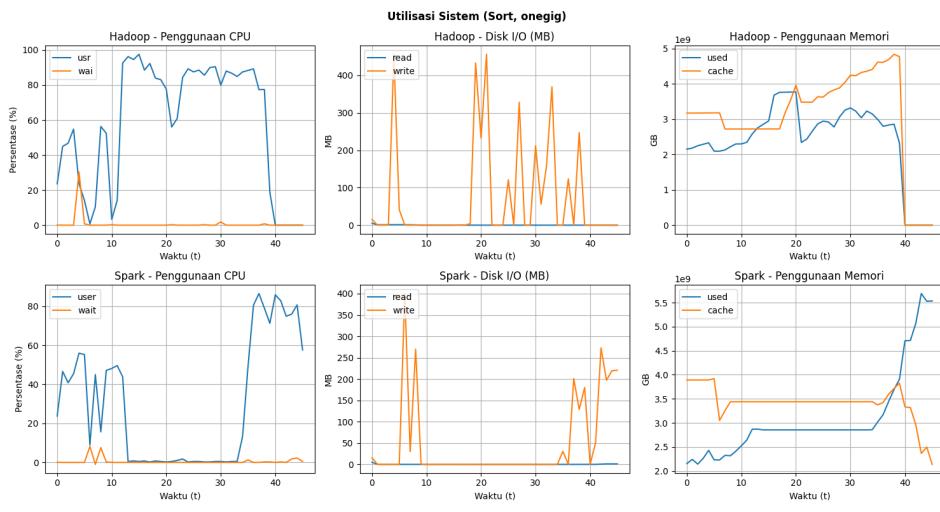


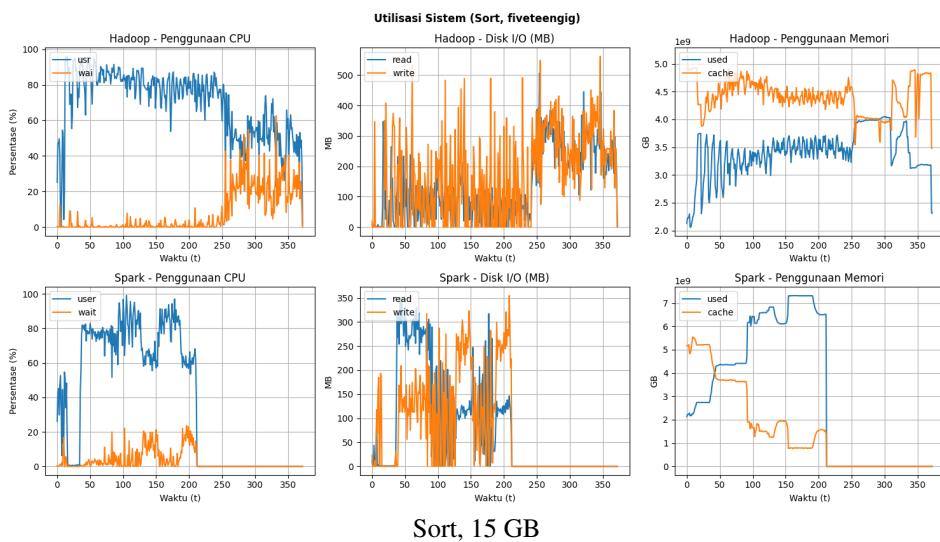
Sort, 5 MB



Sort, 10 MB



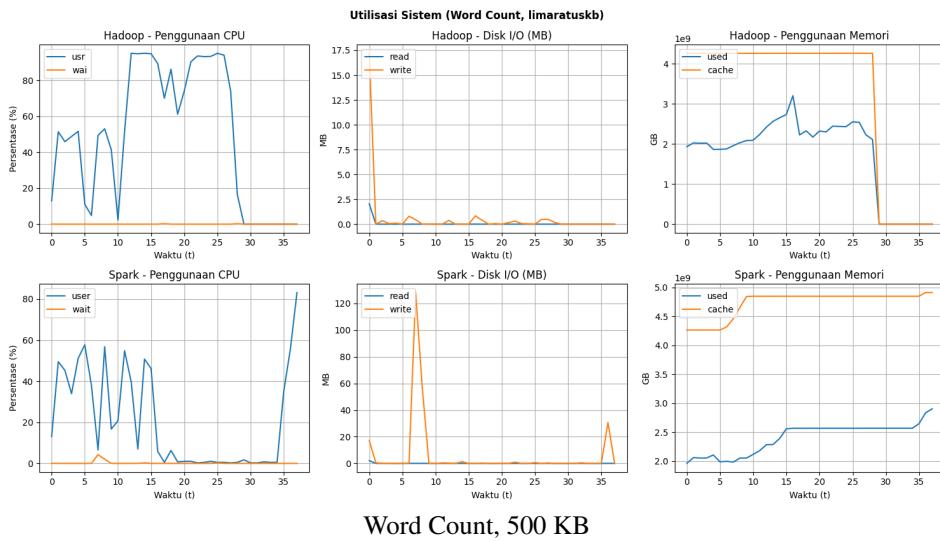
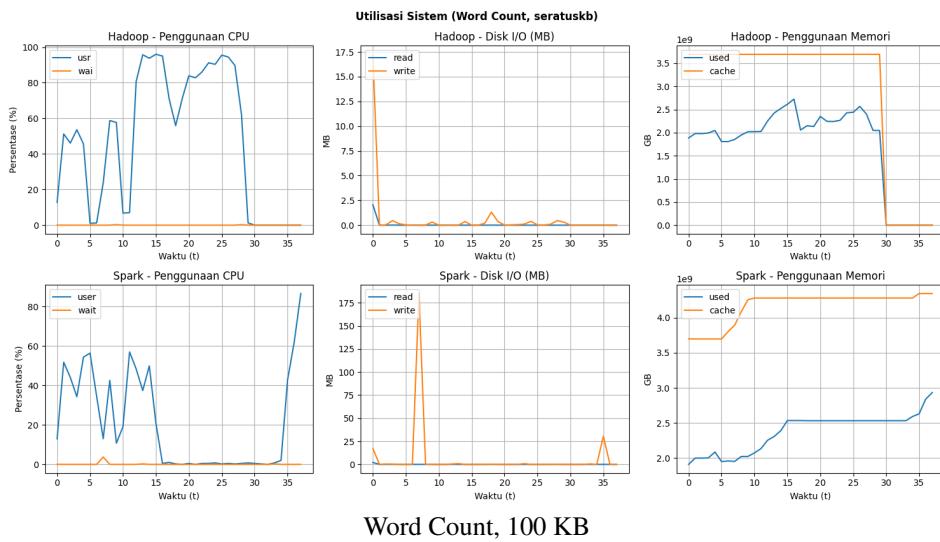


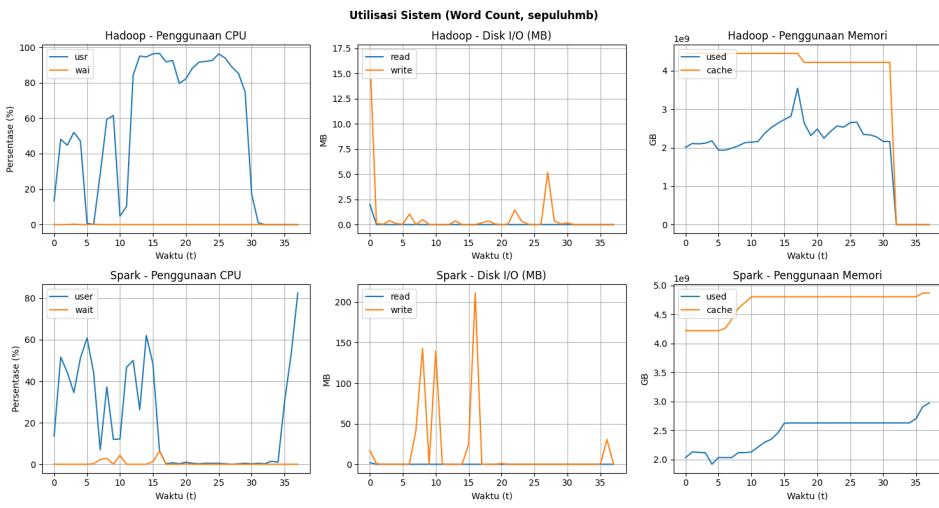
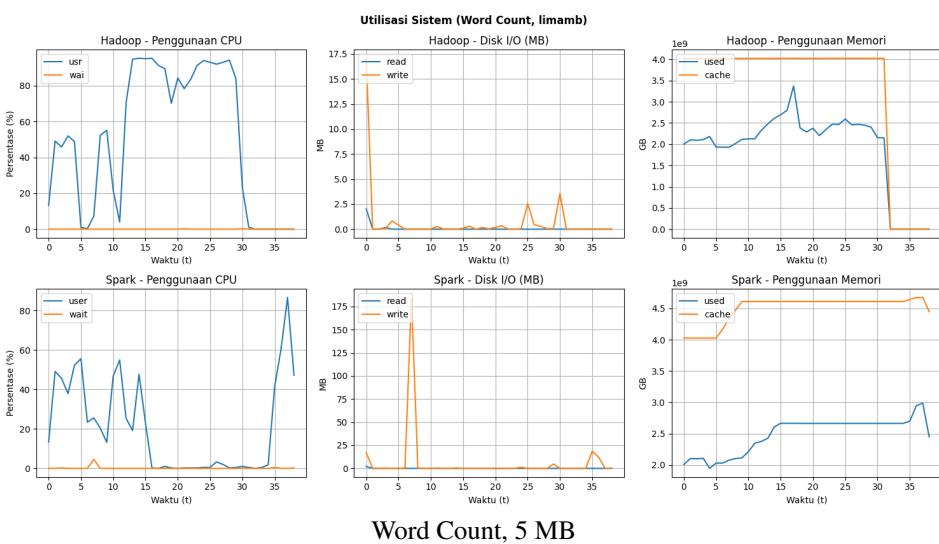
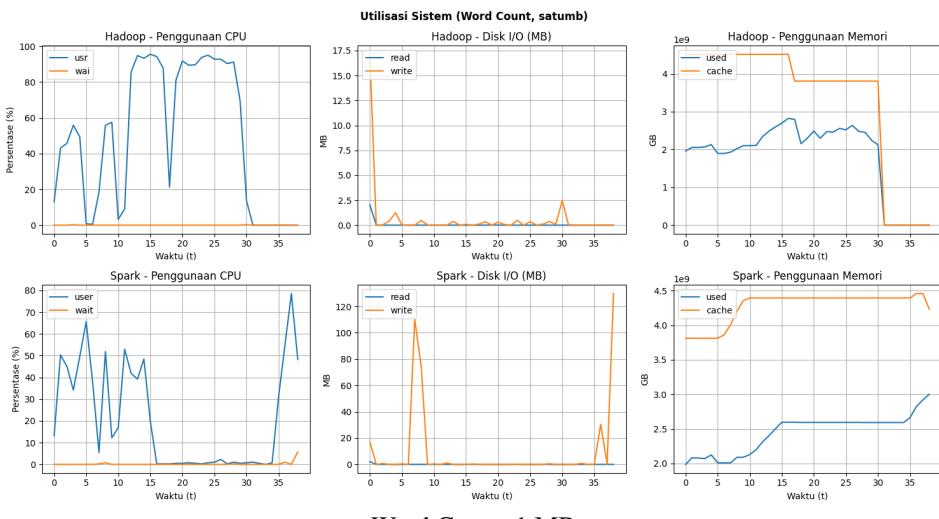


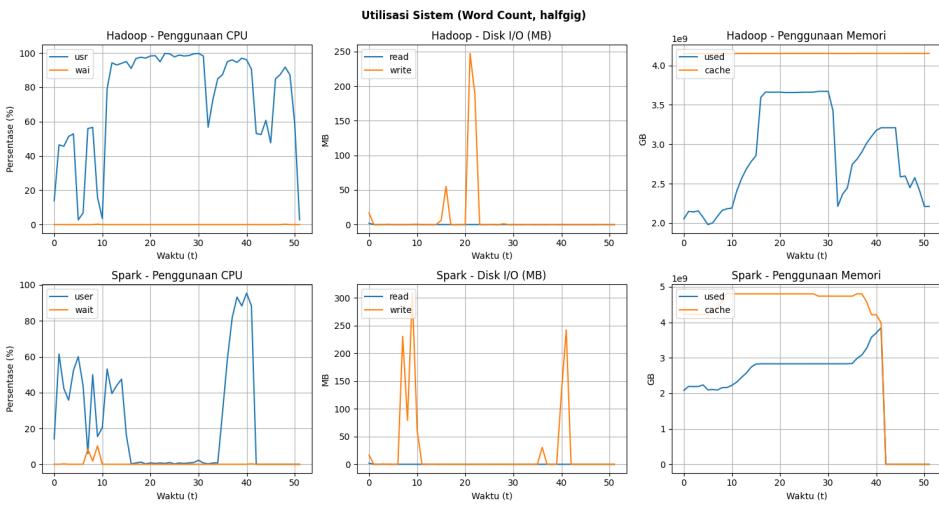
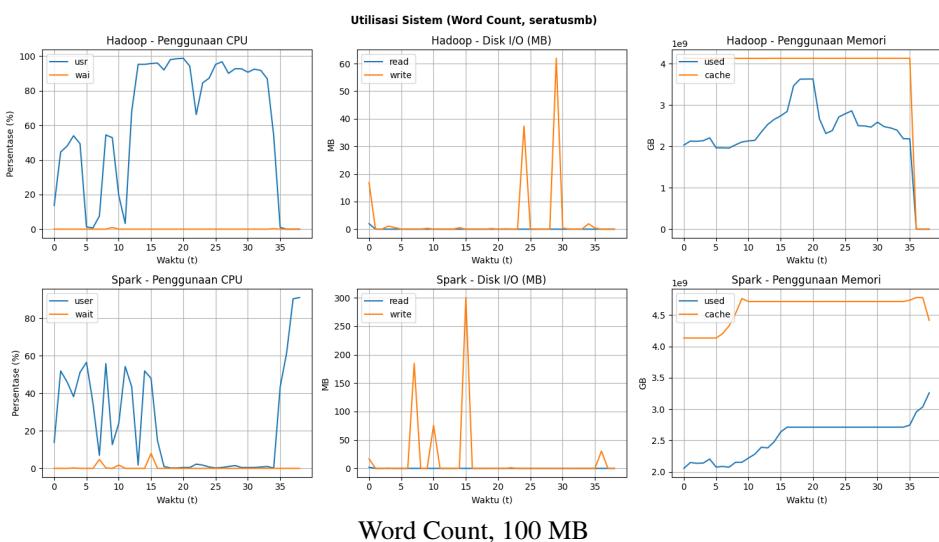
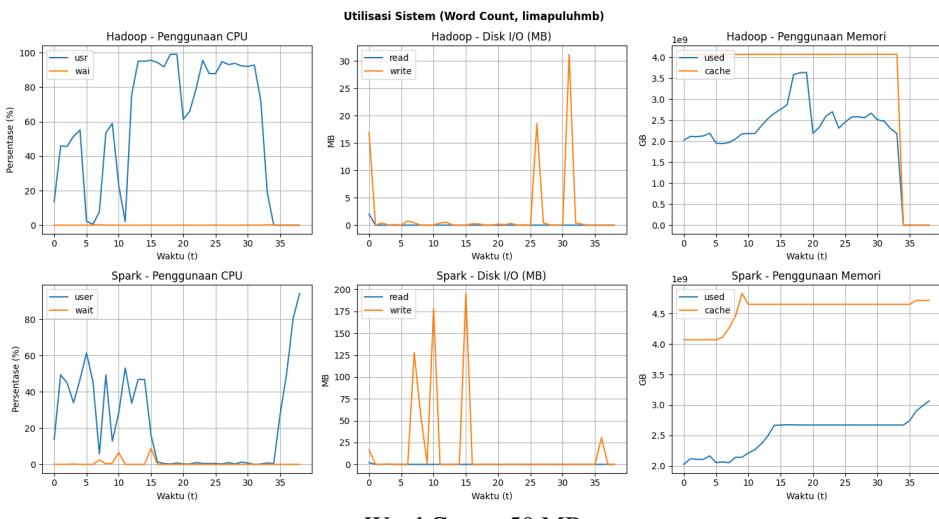
Sort, 15 GB

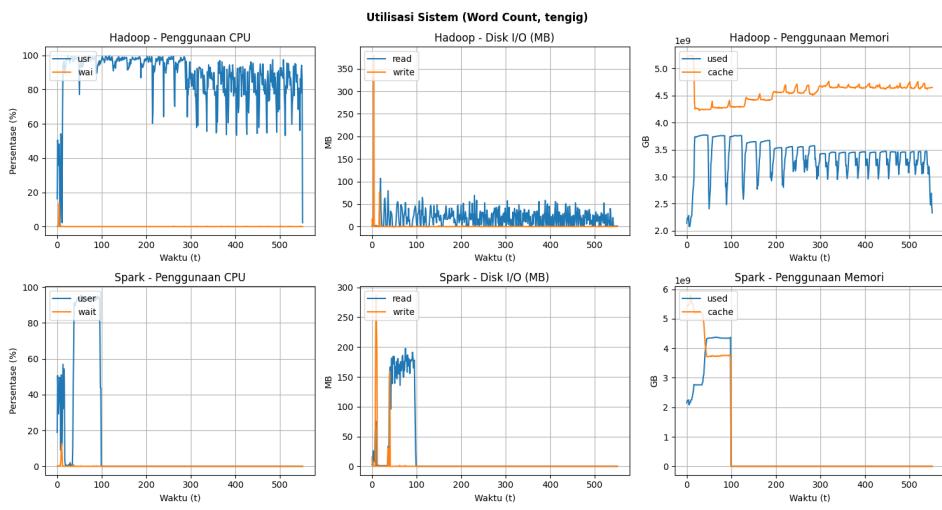
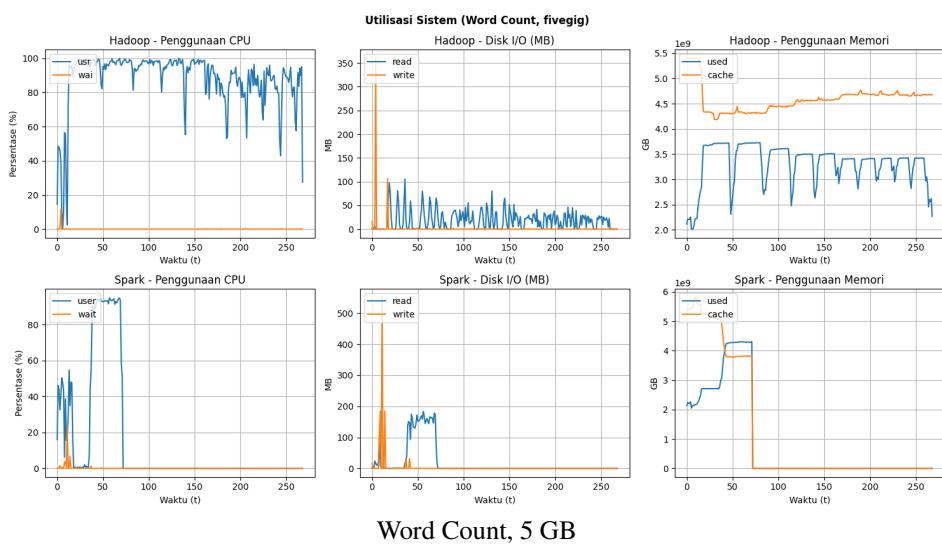
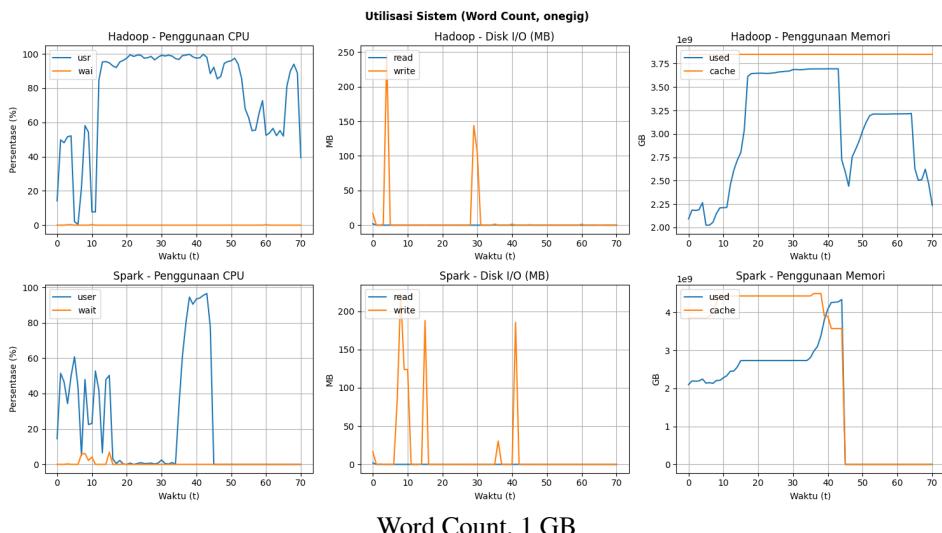
## LAMPIRAN H

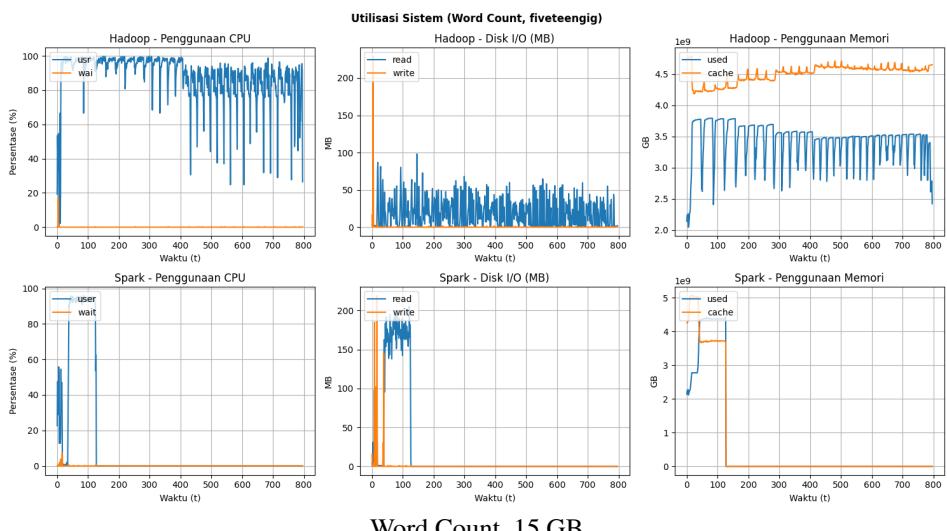
### Visualisasi Utilisasi Sistem Sesuai Input Data (Word Count)











Word Count, 15 GB