



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

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***

SKRIPSI

Diajukan sebagai syarat untuk memperoleh gelar Sarjana

**Dimas Wahyu Saputro
NIM. 120450081**

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

2024

HALAMAN PENGESAHAN

Tugas Akhir Sarjana 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, 3 Juni 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

Penguji I : **Mika Alvionita S, M.Si.**
Penguji II : **Christyan Tamara Nadeak, M.Si.**

Sidang Tugas Akhir :

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 : 3 Juni 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 HiBench dalam Konfigurasi *Pseudo Distributed*

beserta perangkat yang ada (jika diperlukan). Dengan Hak Bebas Royalti Noneksklusif ini Institut Teknologi Sumatera berhak menyimpan, mengalihmedia/format-kan, 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 : 3 Juni 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

Let us do the best, let God do the rest.

HALAMAN PERSEMBAHAN

To all who make our lives worthwhile.

KATA PENGANTAR

Puji syukur penulis ucapkan ke hadirat Allah SWT atas berkat dan rahmat-Nya sehingga skripsi ini dapat terselesaikan dengan baik. Penyusunan skripsi ini banyak mendapat bantuan dan dukungan dari berbagai pihak. Oleh karena itu, dengan penuh kerendahan hati, penulis mengucapkan terima kasih kepada:

1. Orang-orang yang paling istimewa, yaitu kedua orang tua tercinta, Ibu Siti Ervingati dan Bapak Kustriyanto, serta saudara-saudara penulis, Dika, Habib, dan Syifa, yang selalu memberikan doa, semangat, dukungan, dan motivasi sehingga penulis dapat mencapai tahap ini.
2. Bapak Tirta Setiawan, S.Pd., M.Si., selaku Koordinator Program Studi Sains Data Institut Teknologi Sumatera sekaligus Dosen Pembimbing Utama yang telah memberikan arahan, ilmu, motivasi, serta saran kepada penulis.
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 berkuliahan.
5. Abil, Imam, dan Sakul yang selalu memberikan tempat yang aman untuk bercerita dan berbagi canda-tawa. Semoga yang sedang kalian usahakan saat ini dapat terwujud.
6. Alfianri, teman-teman seperbimbingan, teman-teman Sains Data Institut Teknologi Sumatera, dan sahabat-sahabat yang tidak dapat disebutkan satu persatu.

Penulis menyadari bahwa terdapat banyak kekurangan pada penulisan skripsi ini. Oleh karena itu, penulis mengharapkan kritik dan saran yang membangun demi perbaikan. Akhir kata, semoga skripsi ini dapat memberikan manfaat bagi kita semua.

Lampung Selatan, 3 Juni 2024

Dimas Wahyu Saputro

DAFTAR ISI

HALAMAN JUDUL	i
HALAMAN PENGESAHAN	ii
HALAMAN PERNYATAAN ORISINALITAS	iii
HALAMAN PERNYATAAN PERSETUJUAN PUBLIKASI	iv
ABSTRAK	v
ABSTRACT	vi
MOTTO	vii
HALAMAN PERSEMBAHAN	viii
KATA PENGANTAR	ix
DAFTAR ISI	x
DAFTAR GAMBAR	xii
DAFTAR TABEL	xiv
I PENDAHULUAN	1
1.1 Latar Belakang	1
1.2 Rumusan Masalah	3
1.3 Tujuan	3
1.4 Batasan Masalah	3
II LANDASAN TEORI	4
2.1 Tinjauan Pustaka	4
2.2 Konsep <i>Big Data</i>	4
2.3 Statistika Deskriptif	5
2.4 Ekstraksi Fitur Teks (<i>Text Feature Extraction</i>)	6
2.4.1 <i>Bag of Words</i> (BoW)	6
2.4.2 <i>Term Frequency-Inverse Document Frequency</i> (TF-IDF)	7
2.5 Komputasi Awan (<i>Cloud Computing</i>)	8
2.6 <i>Shell Script</i>	8
2.7 <i>CPU Bound</i> dan <i>I/O Bound</i>	9
2.8 MapReduce	10
2.8.1 Apache Hadoop	10
2.8.2 Mode Kerja Hadoop	11
2.8.3 Hadoop Distributed File System (HDFS)	12

2.8.4 Hadoop YARN	12
2.9 Apache Spark	13
2.9.1 Arsitektur Spark	14
2.9.2 Keterbatasan <i>Data Sharing</i> pada MapReduce	14
2.9.3 Solusi <i>Data Sharing</i> dengan Spark RDD	15
2.10 HiBench	16
2.10.1 <i>Data Generation</i> pada <i>Sort</i> dan <i>Word Count</i>	16
2.10.2 Beban Kerja <i>Sort</i>	18
2.10.3 Beban Kerja <i>Word Count</i>	19
2.11 Data Keluaran HiBench dan Dool	20
2.11.1 HiBench <i>Report</i>	20
2.11.2 Dool: <i>System Monitoring</i>	21
III METODOLOGI PENELITIAN	23
3.1 Alur Penelitian	23
3.2 Membangun <i>Virtual Machine</i> di DigitalOcean	24
3.3 Pemasangan dan Konfigurasi Perangkat Lunak	24
3.3.1 Instalasi Perangkat Lunak Prasyarat	25
3.3.2 Instalasi dan Konfigurasi Hadoop	27
3.3.3 Instalasi dan Konfigurasi Spark	27
3.3.4 Instalasi dan Konfigurasi HiBench	28
3.4 Eksperimen	29
3.5 Analisis dan Evaluasi Hasil Eksperimen	32
IV HASIL DAN PEMBAHASAN	34
4.1 Pembangunan <i>Virtual Machine</i> (VM) di DigitalOcean	34
4.2 Pemasangan dan Konfigurasi Perangkat Lunak	34
4.3 Eksperimen	37
4.4 Data Keluaran yang Dihasilkan	38
4.5 Analisis dan Evaluasi Hasil Eksperimen: Kinerja	40
4.5.1 Persebaran Waktu Eksekusi pada Hadoop dan Spark	40
4.5.2 Persebaran <i>Throughput</i> pada Hadoop dan Spark	42
4.6 Analisis dan Evaluasi Hasil Eksperimen: Penggunaan Sumber Daya	46
4.6.1 Penggunaan CPU	46
4.6.2 Utilisasi Sistem	51
4.7 Perbandingan dengan Penelitian Sebelumnya [1]	53
V PENUTUP	54
5.1 Kesimpulan	54

5.2 Saran	54
DAFTAR PUSTAKA	55
LAMPIRAN	59
A Pembuatan <i>Virtual Machine</i> (VM) pada DigitalOcean	60
B Instalasi dan Konfigurasi Perangkat Lunak Prasyarat	62
C Instalasi dan Konfigurasi Hadoop	64
D Instalasi dan Konfigurasi Spark	68
E Instalasi dan Konfigurasi HiBench	69
F Skrip Otomatisasi Eksperimen	71

DAFTAR GAMBAR

Gambar 2.1	<i>CPU-I/O Bound</i> [27]	9
Gambar 2.2	Cara Kerja MapReduce	10
Gambar 2.3	Arsitektur Hadoop	11
Gambar 2.4	Mode Kerja Hadoop [37]	11
Gambar 2.5	Arsitektur HDFS [39]	12
Gambar 2.6	Arsitektur YARN [40]	13
Gambar 2.7	Arsitektur Spark	14
Gambar 2.8	<i>Data Sharing</i> pada MapReduce [42]	15
Gambar 2.9	<i>Data Sharing</i> pada RDD [42]	16
Gambar 2.10	Proses yang Terjadi di HiBench [18]	17
Gambar 2.11	Contoh Kamus pada <i>Random Text Writer</i>	18
Gambar 2.12	Contoh Input dan Output <i>Sort</i>	18
Gambar 2.13	Contoh Input dan Output <i>Word Count</i>	19
Gambar 2.14	Implementasi MapReduce pada Word Count [43]	20
Gambar 2.15	Ilustrasi <i>Throughput</i> [44]	21
Gambar 2.16	Data Keluaran HiBench dan Dool	22
Gambar 3.1	Diagram Alir Penelitian	23
Gambar 3.2	Alur Instalasi Perangkat Lunak Prasyarat	25
Gambar 3.3	Alur Instalasi Perangkat Lunak	26
Gambar 3.4	Alur Instalasi dan Konfigurasi Hadoop	27
Gambar 3.5	Alur Instalasi dan Konfigurasi Spark	28
Gambar 3.6	Alur Instalasi dan Konfigurasi HiBench	28
Gambar 3.7	<i>End-to-end</i> Penelitian	30
Gambar 3.8	Contoh Percobaan	31
Gambar 4.1	Tampilan Dasbor VM DigitalOcean	34
Gambar 4.2	Pengecekan Versi Hadoop	35
Gambar 4.3	Pengecekan Versi Spark	35
Gambar 4.4	Pengecekan <i>Service</i> yang Berjalan (Normal)	36
Gambar 4.5	Pengecekan <i>Service</i> yang Berjalan (Hadoop)	36
Gambar 4.6	Pengecekan <i>Service</i> yang Berjalan (Spark)	37
Gambar 4.7	Data HiBench <i>Report</i>	38
Gambar 4.8	Contoh Data Dool	39
Gambar 4.9	Persebaran Waktu Eksekusi <i>Sort</i> (Hadoop, Spark)	40
Gambar 4.10	Persebaran Waktu Eksekusi <i>Word Count</i> (Hadoop, Spark) .	41

Gambar 4.11 Persebaran <i>Throughput Sort</i> (Hadoop, Spark)	43
Gambar 4.12 Persebaran <i>Throughput Word Count</i> (Hadoop, Spark)	44
Gambar 4.13 Penggunaan CPU (<i>Sort</i>)	49
Gambar 4.14 Penggunaan CPU (<i>Word Count</i>)	50
Gambar 4.15 Utilisasi Sistem (<i>Sort</i>) pada Input Data 15 GB	51
Gambar 4.16 Utilisasi Sistem (<i>Word Count</i>) pada Input Data 15 GB	52

DAFTAR TABEL

Tabel 2.1	Penelitian Terdahulu	4
Tabel 3.1	Konfigurasi Perangkat Keras	24
Tabel 3.2	Perangkat Lunak yang Dibutuhkan	25
Tabel 3.3	Variasi Input Data	31
Tabel 4.1	Konfigurasi HiBench	37
Tabel 4.2	Konfigurasi Spark	37
Tabel 4.3	Statistika Deskriptif Lama Waktu Eksekusi (<i>Sort</i>)	41
Tabel 4.4	Statistika Deskriptif Lama Waktu Eksekusi (<i>Word Count</i>) . .	42
Tabel 4.5	Statistika Deskriptif <i>Throughput</i> (<i>Sort</i>)	44
Tabel 4.6	Statistika Deskriptif <i>Throughput</i> (<i>Word Count</i>)	45
Tabel 4.7	Perbandingan <i>State</i> (<i>Sort</i>)	47
Tabel 4.8	Perbandingan <i>State</i> (<i>Word Count</i>)	48
Tabel 4.9	Rasio Peningkatan Performa Spark-Hadoop [1]	53
Tabel 4.10	Rasio Peningkatan Performa Spark-Hadoop	53

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 lima 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. 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 [8].

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 [9]. Hadoop dan Spark adalah *platform* komputasi *big data* yang populer dan banyak digunakan di seluruh dunia. Pada tahun 2027, Hadoop diperkirakan akan memiliki peningkatan *market size* sebesar USD 341.4 miliar, dari USD 35.3 miliar pada tahun 2024 [10].

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 [11]. 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 [12], sementara Spark menggunakan *Resilient Distributed Datasets* (RDDs) yang bersifat *in-memory*[13].

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 [14]. 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 kinerja (waktu eksekusi, *throughput*), dan penggunaan sumber daya (CPU, memori, *Disk I/O*), penelitian ini bertujuan untuk memberikan wawasan tentang kekuatan dan kelemahan 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.

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, *Disk 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, *Disk 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, *Disk I/O*).
2. Penelitian ini akan fokus pada perbandingan performa 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. Penelitian terdahulu mengenai evaluasi performa Hadoop dan Spark dapat dilihat pada Tabel 2.1.

Tabel 2.1 Penelitian Terdahulu

No	Nama Peneliti	Metode yang Digunakan	Hasil Penelitian	Tahun
1	C. Wibawa, S. Wirawan, M. Mustikasari, dan D. Anggraeni [15]	Konfigurasi empat <i>node</i> . Tiga skenario beban kerja	Spark memiliki kinerja empat kali lebih cepat dibandingkan Hadoop	2022
2	N. Ahmed, A. L. C. Barczak, T. Susnjak, dan M. A. Rashid [16]	HiBench. Konfigurasi sembilan <i>node</i> . <i>Word Count</i> and <i>TeraSort</i>	Spark dua kali lebih cepat pada beban kerja Word Count, 14 kali pada <i>TeraSort</i>	2020
3	Yassir Samadi, Mostapha Zbakh, Claude Tadonki [1]	HiBench. Konfigurasi <i>Pseudo Distributed</i> . Sembilan beban kerja	Spark memiliki kinerja empat kali lebih cepat dibandingkan Hadoop	2018
4	Satish Gopalani, Rohan Arora [17]	K-Means. Konfigurasi dua <i>node</i>	Spark memiliki kinerja dua kali lebih cepat dibandingkan dengan Hadoop	2015

2.2 Konsep *Big Data*

Big data 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 berteknologi.

serta algoritma yang lebih canggih. Pendekatan teknologi yang sering digunakan oleh *business intelligence* biasanya tidak dapat lagi efisien jika digunakan.

Big data 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). 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. Sementara itu, di sektor keuangan, *big data* membantu dalam analisis risiko, deteksi penipuan, dan personalisasi layanan untuk pelanggan.

2.3 Statistika Deskriptif

Statistika deskriptif merupakan cabang statistika yang fokus pada pengorganisasian, penyajian, dan pengikhtisan data [22]. Informasi yang diperoleh dari statistika deskriptif berguna untuk mendapatkan gambaran umum mengenai data dan untuk memahaminya secara lebih mendalam. Beberapa ukuran statistika deskriptif yang umum digunakan antara lain,

1. Maksimum (Max), merupakan nilai tertinggi dalam suatu kumpulan data.
2. Minimum (Min), merupakan nilai terendah dalam suatu kumpulan data.
3. Rata-rata (Mean), merupakan nilai tengah dari suatu kumpulan data.

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n} \quad (2.1)$$

Keterangan:

- \bar{x} adalah mean
- x_i adalah nilai data ke-i
- n adalah jumlah total data

- Standar Deviasi (σ), merupakan persebaran data pada suatu sampel untuk melihat seberapa jauh atau seberapa dekat nilai data dengan rata-ratanya. Semakin besar standar deviasi, semakin tersebar data dari mean.

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n}} \quad (2.2)$$

Keterangan:

- σ adalah standar deviasi
- x_i adalah nilai data ke-i
- \bar{x} adalah mean
- n adalah jumlah total data

2.4 Ekstraksi Fitur Teks (*Text Feature Extraction*)

Penerapan algoritma *sort* dan *word count* dapat dilihat pada ekstraksi fitur teks. Ekstraksi fitur teks bertujuan untuk mengurangi kompleksitas data (atau yang sering disebut juga *Data Dimensionality*) namun tetap menyimpan sebanyak mungkin informasi yang paling relevan. Data mentah yang mengandung banyak fitur tidak relevan 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.

2.4.1 *Bag of Words* (BoW)

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

1. Tokenisasi, yaitu teks dipecah menjadi kata-kata individual (*token*).
2. Pembuatan kosakata yang berisi daftar unik dari semua token yang ada dalam seluruh kumpulan dokumen dibuat.
3. Penghitungan Kata (*Word Count*), yaitu menghitung frekuensi kemunculan setiap kata dalam dokumen.
4. Representasi Vektor, yaitu 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.

2.4.2 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), yaitu mengukur seberapa sering suatu kata muncul dalam dokumen.
2. *Inverse Document Frequency* (IDF), yaitu 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.

Nilai TF-IDF akan mencerminkan pentingnya kata dalam dokumen dan koleksi dokumen. Rumus umum untuk TF-IDF sebagai berikut,

$$w_{ij} = tf_{ij} \times \log\left(\frac{N}{df_i}\right) \quad (2.3)$$

Keterangan:

- w_{ij} adalah bobot tf-idf untuk kata i dalam dokumen j ;
- tf_{ij} adalah frekuensi kemunculan kata i dalam dokumen j dibagi dengan total jumlah kata dalam dokumen j ;
- N adalah total jumlah dokumen dalam korpus;
- df_i adalah jumlah dokumen dalam korpus yang mengandung kata i .

Sebagai contoh, terdapat kata *British* yang muncul lima kali dalam sebuah dokumen yang berisi seratus kata. Dengan korpus yang berisi empat dokumen, dimana dua dokumen menyebutkan kata *British*, TF-IDF dapat dihitung sebagai berikut,

$$w_{British} = \frac{5}{100} \times \log\left(\frac{4}{2}\right) = 0.015$$

TF-IDF meningkat seiring dengan peningkatan frekuensi kemunculan kata dalam dokumen, tetapi menurun seiring dengan peningkatan jumlah dokumen lain dalam korpus yang juga mengandung kata tersebut. Variasi dari skema pembobotan TF-IDF sering digunakan oleh mesin pencari sebagai alat utama dalam menilai dan mengurutkan relevansi dokumen terhadap *query* pengguna.

2.5 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 aplikasi Google Mail, Microsoft One Drive, dan Siakad Itera.

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 Hadoop atau Spark. Salah satu komputasi awan yang dapat diandalkan adalah DigitalOcean. DigitalOcean didirikan pada tahun 2012 untuk memenuhi kebutuhan pengembang demi 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, serta memilih sistem operasi apa yang akan digunakan.

2.6 *Shell Script*

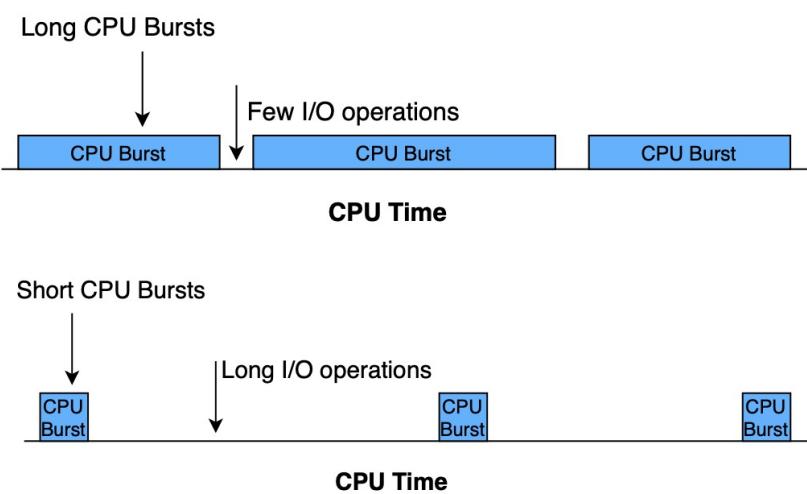
Shell script merupakan rangkaian perintah yang dieksekusi dalam lingkungan sistem operasi Unix atau *Unix-like* [26]. *Shell script* memungkinkan pengguna untuk mengotomatiskan tugas-tugas rutin, melakukan pemrosesan file, dan membangun aplikasi yang kompleks dengan menggunakan perintah-perintah shell. *Shell script* umumnya ditulis menggunakan bahasa pemrograman shell, seperti Bash (Bourne 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 tersebut dapat dijadwalkan untuk berjalan secara otomatis sehingga proses pencadangan dapat dilakukan tanpa campur tangan manusia secara berkala.

2.7 CPU Bound dan I/O Bound

Terdapat dua jenis beban kerja pada pemrosesan data berdasarkan karakteristik dan kebutuhan sumber daya yang berbeda, yaitu *CPU-bound* dan *I/O-bound*. Proses *CPU-bound* (Gambar 2.1, bagian atas) sangat bergantung pada kemampuan pemrosesan data oleh *Central Processing Unit* (CPU). Proses ini menghabiskan sebagian besar waktunya dalam menjalankan instruksi CPU dan jarang berinteraksi dengan sistem I/O (*Input/Output*). Sebaliknya, proses *I/O-bound* (Gambar 2.1, bagian bawah) lebih banyak menghabiskan waktu dalam menunggu operasi I/O, seperti akses *disk*, dan jaringan. CPU dalam proses ini hanya digunakan sesaat untuk memproses data yang baru saja diperoleh dari I/O.

Proses *CPU-bound* memiliki interval waktu yang panjang untuk menjalankan instruksi CPU, dan jarang melakukan operasi I/O, yang dikenal sebagai *CPU bursts* yang panjang dan jarang. Sementara itu, proses *I/O-bound* memiliki interval waktu yang pendek untuk menjalankan instruksi CPU, dan sering melakukan operasi I/O, sehingga memiliki *CPU bursts* yang pendek dan sering.

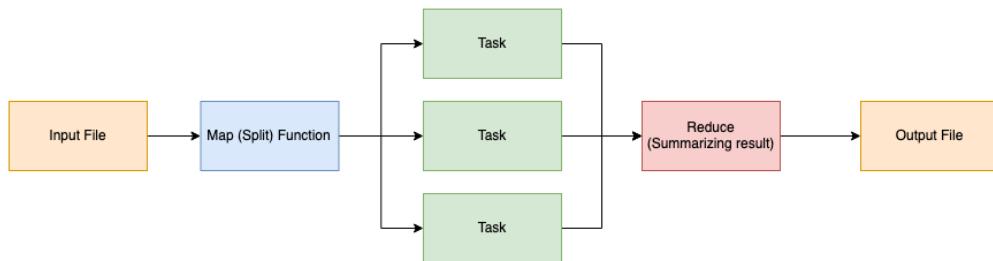


Gambar 2.1 CPU-I/O Bound [27]

2.8 MapReduce

MapReduce adalah model pemrograman dan implementasi teknik pemrosesan data berukuran besar yang pertama kali dipopulerkan oleh Google pada tahun 2004[28]. MapReduce menawarkan pemrosesan data yang dapat diandalkan serta *fault-tolerant manner* (tahan terhadap kesalahan). MapReduce berjalan secara paralel dan berada pada lingkungan komputasi terdistribusi [29]. Model ini mengadopsi arsitektur tersentraliasi, yaitu satu *node* berperan sebagai *master* dan *node* yang lain berperan sebagai *workers* atau *slave* [30], [31]. *Master node* bertanggung jawab untuk melakukan penjadwalan kerja, dan *slave node* berperan untuk menjalankan eksekusi kerja.

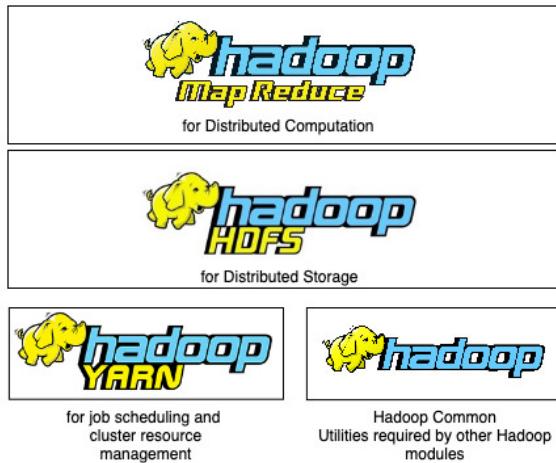
MapReduce terdiri dari fungsi *Map* dan fungsi *Reduce*, sesuai dengan Gambar 2.2 [32]. Kedua fungsi ini tersebar di seluruh *slave node* yang terhubung dalam klaster dan berjalan secara paralel. Fungsi *Map* berperan untuk membagi masalah besar menjadi masalah yang lebih kecil dan mendistribusikannya ke *slave node*. Hasil pemrosesan dari *slave node* akan dikumpulkan oleh *master node* melalui fungsi *Reduce*. Hasil dari proses *Reduce* yang akan dikirimkan sebagai keluaran akhir proses MapReduce.



Gambar 2.2 Cara Kerja MapReduce

2.8.1 Apache Hadoop

Apache Hadoop adalah perangkat lunak sumber terbuka yang ditulis dengan bahasa pemrograman Java untuk pemrosesan dan penyimpanan data menggunakan komputasi terdistribusi [33]. Hadoop dapat diinstalasi pada satu *node* komputer, atau ratusan *node* komputer yang digabungkan dalam sebuah klaster [34]. 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 [35]. Arsitektur Hadoop secara umum dapat dilihat pada Gambar 2.3.

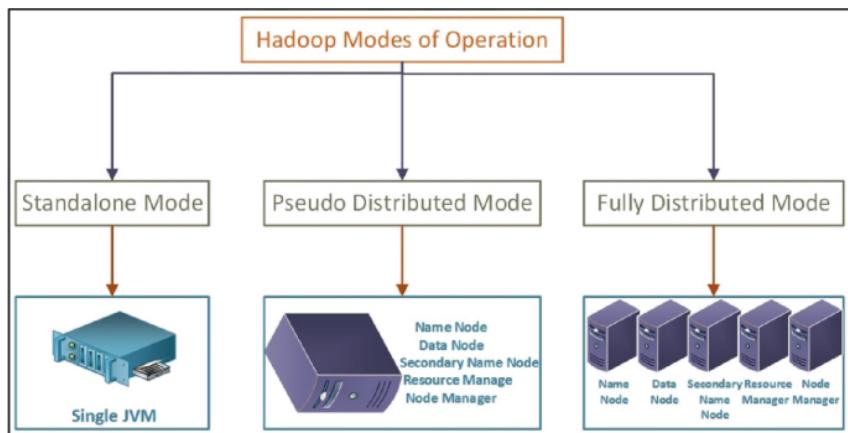


Gambar 2.3 Arsitektur Hadoop

2.8.2 Mode Kerja Hadoop

Hadoop dapat dijalankan dalam tiga mode operasi yang berbeda yaitu *standalone*, *pseudo-distributed*, dan *fully distributed* [36]. Dalam *standalone mode*, semua proses Hadoop berjalan pada satu *node* tunggal menggunakan sistem berkas lokal tanpa memerlukan konfigurasi tambahan pada Hadoop seperti pada Gambar 2.4.

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

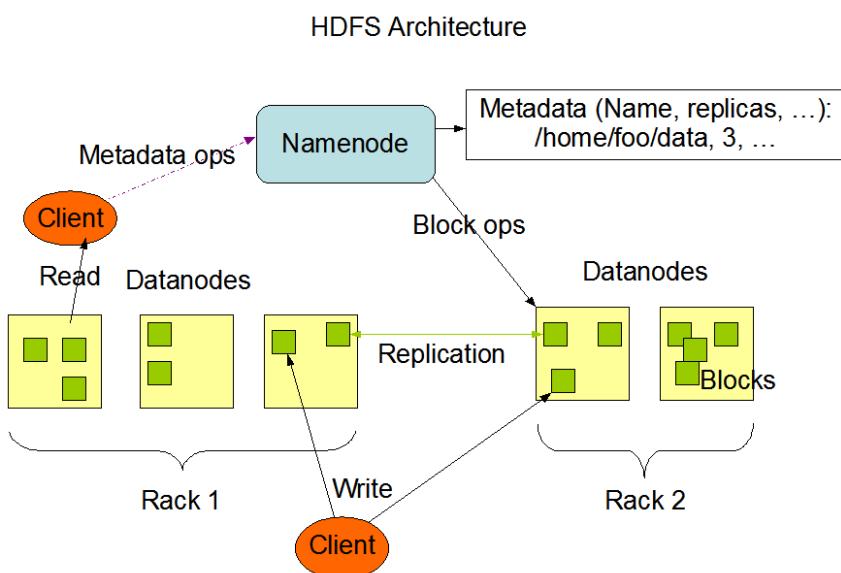


Gambar 2.4 Mode Kerja Hadoop [37]

2.8.3 Hadoop Distributed File System (HDFS)

Hadoop Distributed File System adalah sistem file terdistribusi yang dikembangkan sebagai bagian dari Hadoop [38]. 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.

Secara struktur, HDFS terdiri dari *Name Node* sebagai *node master* yang mengelola *metadata* dan *namespace*, serta *Data Node* sebagai *node slave* yang bertugas menyimpan data sebenarnya dalam bentuk blok seperti pada Gambar 2.5. Berkas pada HDFS dipartisi menjadi satu atau lebih blok berukuran 64 MB atau 128 MB, kemudian didistribusikan dan disimpan di beberapa *Data Node*. Setiap blok direplikasi di *Data Node* yang berbeda untuk toleransi kesalahan. Replikasi blok di *node/rack* yang berbeda juga meningkatkan ketersediaan HDFS.



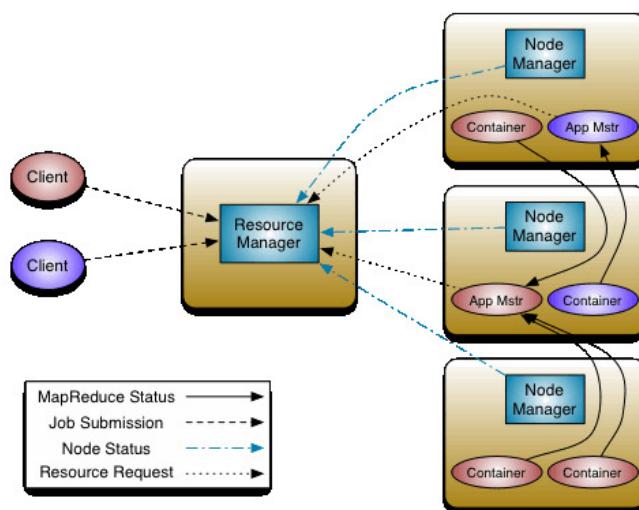
Gambar 2.5 Arsitektur HDFS [39]

2.8.4 Hadoop YARN

Hadoop YARN (Yet Another Resource Negotiator) adalah manajer sumber daya dan sistem penjadwalan untuk klaster 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 klaster 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*.

Struktur utama YARN terdiri dari *Resource Manager* yang bertugas mengoordinasikan alokasi sumber daya di seluruh klaster, dan *Node Manager* yang berjalan di setiap *node* untuk mengawasi penggunaan sumber daya dan mengelola *container* tempat aplikasi dijalankan. *Application Master* adalah komponen khusus yang bertanggung jawab untuk negosiasi sumber daya dengan *Resource Manager* dan bekerja dengan *Node Manager* untuk menjalankan dan memantau *tasks* seperti pada Gambar 2.6.



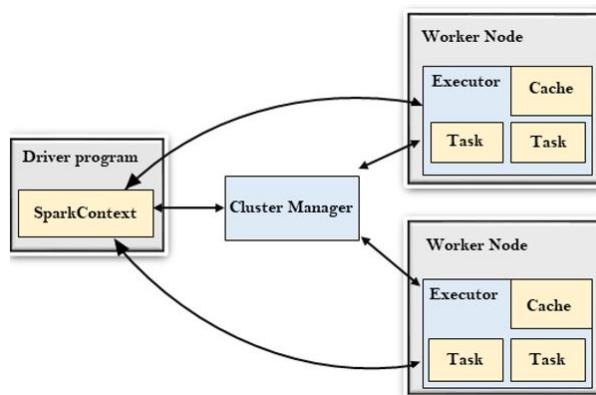
Gambar 2.6 Arsitektur YARN [40]

2.9 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. 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.

2.9.1 Arsitektur Spark

Arsitektur Spark dirancang untuk pemrosesan data terdistribusi yang efisien dan cepat seperti pada Gambar 2.7. 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 penyimpanan data, menjadikannya solusi ideal untuk berbagai kasus penggunaan data besar.



Gambar 2.7 Arsitektur Spark

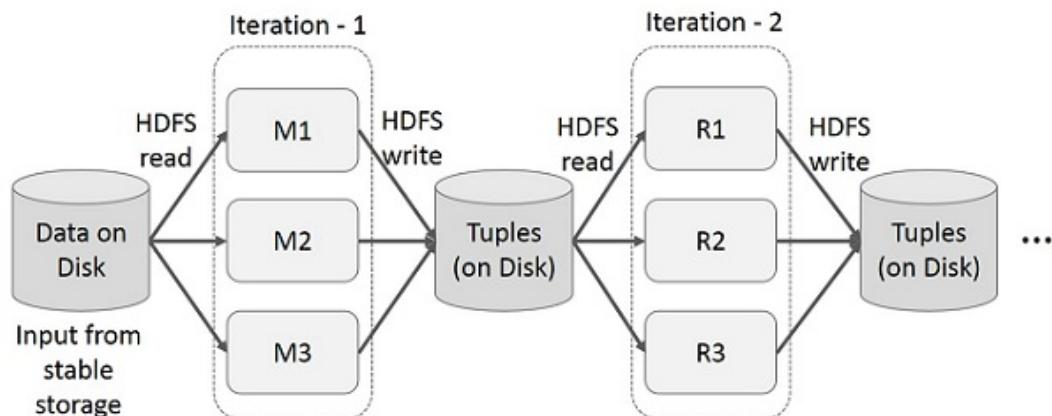
2.9.2 Keterbatasan *Data Sharing* pada MapReduce

MapReduce, sebagai kerangka kerja pemrosesan data terdistribusi, mengandalkan sistem penyimpanan eksternal yang stabil, seperti HDFS, untuk berbagi data antar tugas (*job*). Hal ini mengakibatkan inefisiensi 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, dimana 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 2.8. Akibatnya, aplikasi iteratif pada MapReduce cenderung lambat dan tidak efisien.



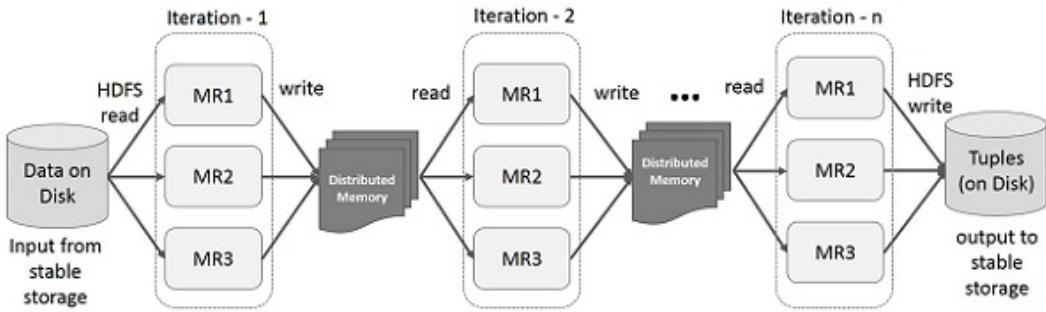
Gambar 2.8 Data Sharing pada MapReduce [42]

2.9.3 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*, dimana 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 2.9. Dengan demikian, Spark RDD memungkinkan eksekusi aplikasi iteratif yang jauh lebih cepat dan efisien dibandingkan dengan MapReduce.



Gambar 2.9 Data Sharing pada RDD [42]

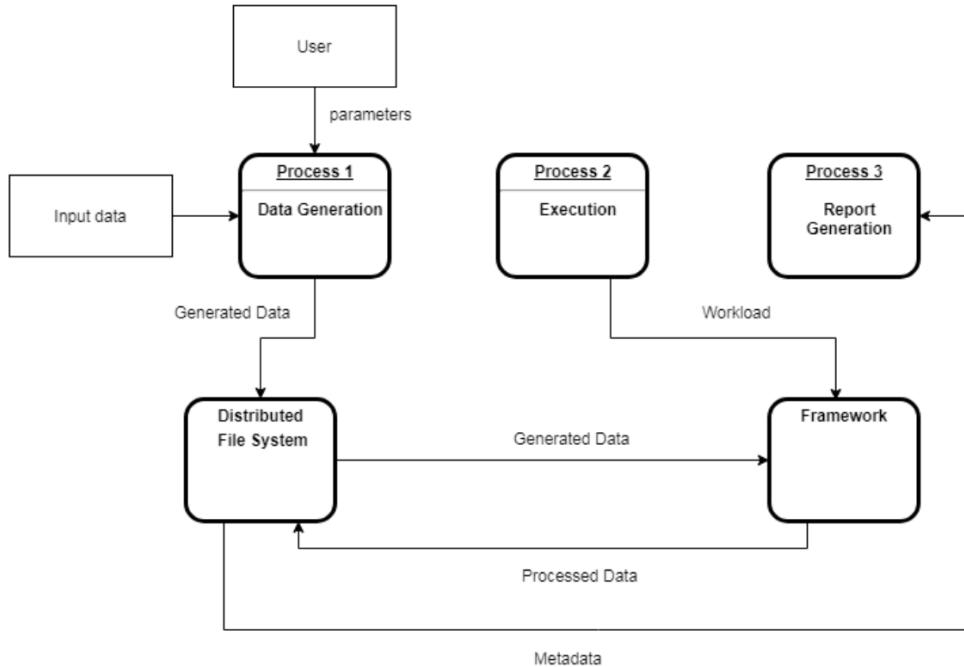
2.10 HiBench

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 2.10.

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 Time* dan *Throughput*.

2.10.1 Data Generation pada Sort dan Word Count

Data generation merupakan tahapan krusial dalam *benchmark* menggunakan HiBench, khususnya untuk beban kerja *sort* dan *word count*. Tahapan ini bertanggung jawab untuk membentuk data acak yang akan diproses oleh kedua beban kerja tersebut. Tujuannya adalah untuk menyimulasikan skenario nyata dengan data masukan yang bervolume besar dan beragam.



Gambar 2.10 Proses yang Terjadi di HiBench [18]

Pada HiBench, skrip persiapan seperti pada Algoritma 2.1 berperan penting dalam menyiapkan data beban kerja. Skrip ini mengeksekusi program *Random Text Writer* yang terdapat dalam paket Hadoop. *Random Text Writer* 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 tersebut.

Algoritma 2.1 Skrip HiBench pada Tahap *Data Generation*

```

1 #!/bin/bash
2
3 START_TIME=`timestamp`
4 run_hadoop_job ${HADOOP_EXAMPLES_JAR} randomtextwriter \
5   -D mapreduce.randomtextwriter.totalbytes=${DATASIZE} \
6   -D mapreduce.randomtextwriter.bytespermap=$(( ${DATASIZE} / \
7   ${NUM_MAPS} )) \
8   -D mapreduce.job.maps=${NUM_MAPS} \
9   -D mapreduce.job.reduces=${NUM_RED} \
10  ${INPUT_HDFS}
11 END_TIME=`timestamp`
```

Dalam penerapannya, *Random Text Writer* akan menghitung jumlah total *map task* yang diperlukan berdasarkan konfigurasi dan status klaster Hadoop. Setiap *map task* akan menulis data hingga mencapai jumlah bita yang telah dikonfigurasi, menghasilkan kata acak sesuai dengan panjang yang telah ditentukan. Kata acak tersebut berasal dari kamus (*dictionary*) yang sebelumnya sudah didefinisikan seperti pada Gambar 2.11.

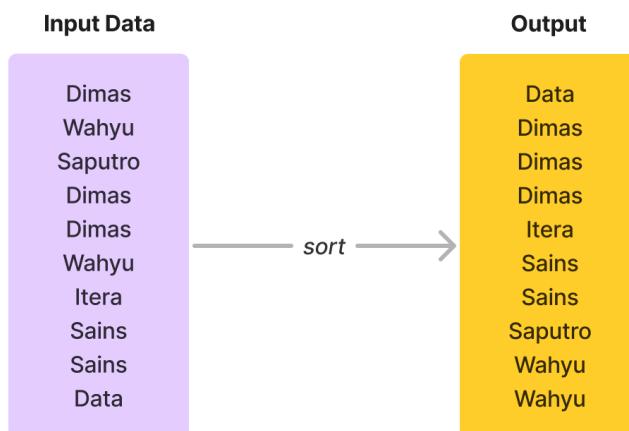
Jika diberikan input yang pasti, misalnya 100 KB, *Random Text Writer* menghasilkan keluaran dengan ukuran tersebut tanpa pemotongan kata karena *mapper* menghasilkan kata acak dalam kata lengkap dan menghitung panjang kata-kata tersebut sebelum menulisnya.

```
/**  
 * A random list of 1000 words from /usr/share/dict/words  
 */  
private static String[] words = {  
    "diurnalness", "Homoiousian",  
    "spiranthic", "tetragynian",  
    "silverhead", "ungreat",  
    "undeterring", "antiscolic",  
    "schoolmasterism", "nonuple",  
    // ...  
}
```

Gambar 2.11 Contoh Kamus pada *Random Text Writer*

2.10.2 Beban Kerja *Sort*

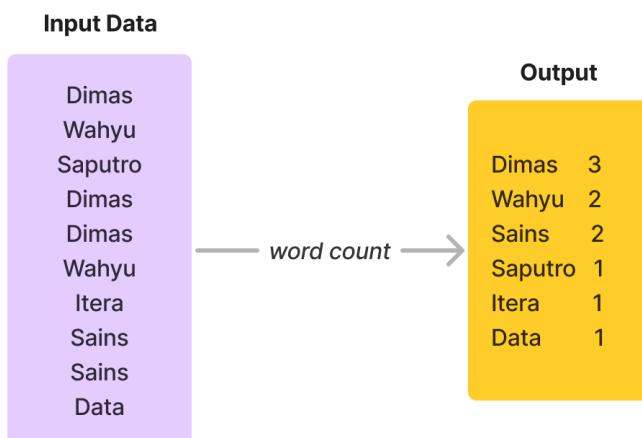
Sort adalah algoritma yang umum digunakan untuk mengurutkan data berdasarkan kriteria tertentu. Algoritma ini menerima data dalam bentuk acak sebagai masukan, dan menghasilkan data yang terurut sebagai keluaran seperti pada Gambar 2.12. Data masukan dan keluaran memiliki ukuran yang sama, sehingga beban 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 2.12 Contoh Input dan Output *Sort*

2.10.3 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, masukannya berupa berkas teks dan keluarannya berupa pasangan kata-kata dan jumlah kemunculannya. Beban kerja *word count* akan menghasilkan data keluaran yang lebih kecil dari pada data masukan seperti pada Gambar 2.13. 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, perilakunya akan tetap sama bahkan pada klaster yang lebih besar.



Gambar 2.13 Contoh Input dan Output *Word Count*

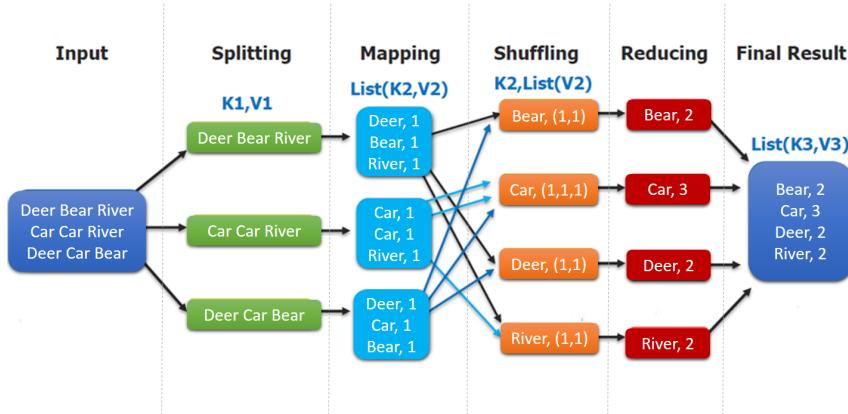
Implementasi MapReduce pada *word count* dapat dilihat pada Gambar 2.14. Pada proses MapReduce, data masukan akan melalui beberapa tahapan pemrosesan. Pertama, data akan dipecah menjadi bagian-bagian yang lebih kecil pada proses pemecahan data masukan (*splitting*). Dalam kasus Hadoop MapReduce, data idealnya akan dipecah menjadi beberapa blok berukuran maksimal 128 MB.

Kemudian, blok data tersebut akan diproses lebih lanjut pada tahap pemetaan (*mapping*). Pemetaan merupakan salah satu tahapan terpenting dalam MapReduce. Pada tahap ini, blok data yang sudah dipecah akan diproses untuk menghasilkan pasangan kunci-nilai (*key-value pairs*) sementara, seperti pada contoh kasus *word count* 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*) dimana 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 *word count*, pasangan *Bear:1* dan *Bear:1* akan dijumlahkan menjadi *Bear:2* oleh proses pengurangan.



Gambar 2.14 Implementasi MapReduce pada Word Count [43]

2.11 Data Keluaran HiBench dan Dool

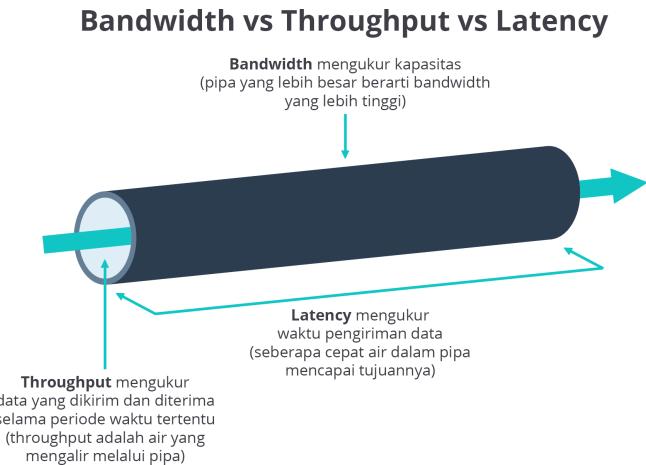
Diagram pada Gambar 2.16 mengilustrasikan data keluaran dari dua alat, yaitu HiBench dan Dool, yang digunakan untuk mengevaluasi performa sistem. HiBench berfokus pada pengukuran kinerja keseluruhan dari suatu beban kerja (*benchmark*), sementara Dool memberikan pemantauan sistem yang terperinci secara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

2.11.1 HiBench Report

HiBench menghasilkan laporan yang mencakup dua metrik utama, yaitu

- Waktu Eksekusi (Execution Time):** Waktu eksekusi suatu proses dapat dihitung dengan mengambil perbedaan antara waktu akhir dan waktu awal dari proses tersebut dalam format *timestamp*. *Timestamp* yang digunakan adalah dalam bentuk *Unix time*, yaitu representasi waktu sebagai jumlah detik yang telah berlalu sejak 1 Januari 1970 UTC. Untuk mendapatkan waktu eksekusi, pertama-tama catat waktu awal proses menggunakan *Timestamp Unix*. Setelah proses selesai, catat waktu akhirnya dengan cara yang sama. Selanjutnya, kurangi waktu awal dari waktu akhir untuk memperoleh durasi eksekusi dalam satuan 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. Ilustrasi *Throughput* terlihat seperti pada Gambar 2.15



Gambar 2.15 Ilustrasi *Throughput* [44]

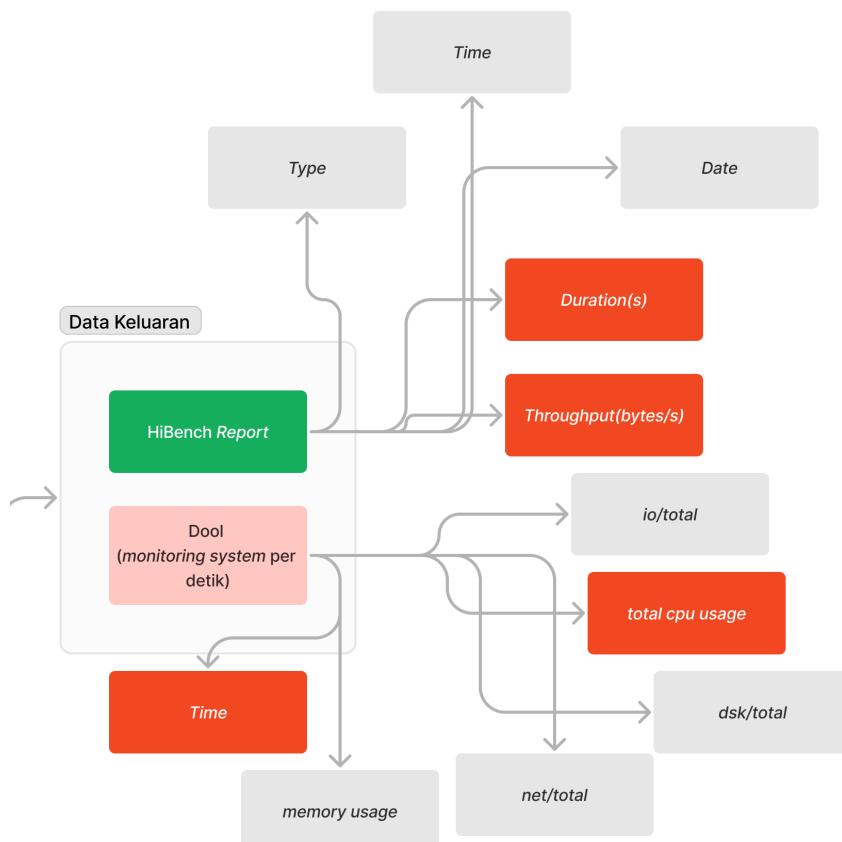
2.11.2 Dool: *System Monitoring*

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 data.
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 detik.
5. ***total cpu usage***: Persentase penggunaan CPU secara keseluruhan.
 - (a) ***cpu/user***: Persentase waktu CPU yang digunakan oleh proses-proses di ruang pengguna (*user space*). Ini mencerminkan aktivitas dari aplikasi-aplikasi dan proses-proses yang dijalankan oleh pengguna.
 - (b) ***cpu/wait***: Persentase waktu CPU yang dihabiskan untuk menunggu operasi I/O selesai. Angka yang tinggi pada metrik ini bisa menunjukkan adanya *bottleneck* pada disk atau jaringan yang menyebabkan CPU harus menunggu data tersedia.
6. ***disk/total***: Total aktivitas *disk*, mencakup baca dan tulis, diukur dalam bita per detik.
 - (a) ***disk/read***: Jumlah data yang dibaca dari *disk*, diukur dalam bita per detik. Metrik ini penting untuk mengidentifikasi seberapa banyak beban pembacaan yang diterima oleh sistem penyimpanan.

- (b) ***disk/write***: Jumlah data yang ditulis ke *disk*, diukur dalam bita per detik. Metrik ini menunjukkan seberapa banyak data yang sedang ditulis ke penyimpanan, yang bisa mempengaruhi kinerja sistem jika terlalu tinggi.
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.

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



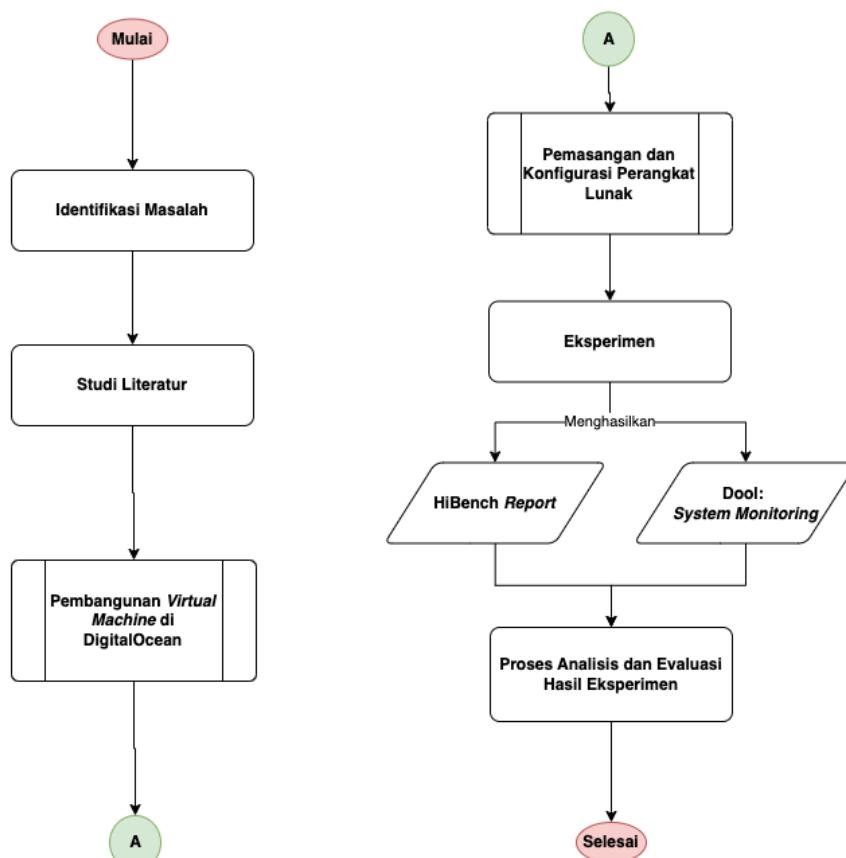
Gambar 2.16 Data Keluaran HiBench dan Dool

BAB III

METODOLOGI PENELITIAN

3.1 Alur Penelitian

Penelitian ini terdapat enam tahapan utama yang ditunjukkan pada Gambar 3.1. Langkah awal yang dilakukan adalah melakukan identifikasi masalah, yaitu proses mencari, dan menghimpun permasalahan yang akan diselesaikan. Selanjutnya adalah studi literatur. Studi literatur adalah tahapan untuk mencari solusi dari permasalahan yang sebelumnya sudah didefinisikan. Kemudian, penelitian ini akan dilanjutkan pada tahap membangun *virtual machine* di DigitalOcean. *Virtual Machine* tersebut dapat dihentikan atau dihapus saat tidak lagi diperlukan. Ketika infrastruktur sudah siap digunakan, penelitian dilanjutkan ke tahap pemasangan perangkat lunak, seperti Hadoop, Spark, dan HiBench. Selanjutnya dilakukan eksperimen pada beban kerja *sort* dan *word count*. Hasil dari eksperimen akan digunakan untuk proses analisis dan evaluasi.



Gambar 3.1 Diagram Alir Penelitian

3.2 Membangun *Virtual Machine* di DigitalOcean

Konfigurasi perangkat keras merupakan aspek penting dalam mengevaluasi kinerja aplikasi *big data* berbasis Hadoop dan Spark. Penelitian ini memerlukan penggunaan mesin virtual, Droplets pada DigitalOcean, 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 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.

Tabel 3.1 Konfigurasi Perangkat Keras

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

3.3 Pemasangan dan Konfigurasi Perangkat Lunak

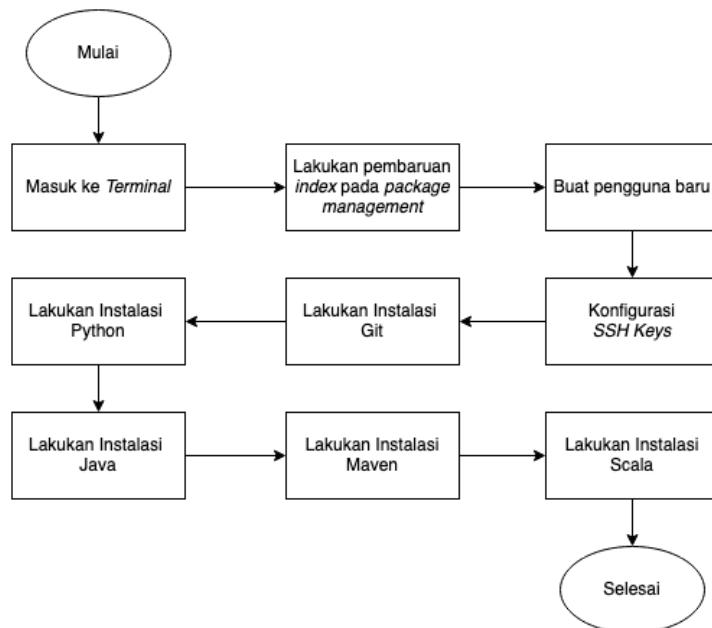
Pemasangan dan konfigurasi perangkat lunak merupakan hal yang krusial dalam penelitian ini. Perangkat lunak yang diperlukan ditunjukkan pada Tabel 3.2. Selanjutnya, alur instalasi perangkat lunak dalam penelitian ini dapat dilihat pada Gambar 3.3. Terdapat tiga bagian utama, yaitu *prerequisites* (perangkat lunak prasyarat) ditandai dengan warna biru, alat penyimpanan dan pemrosesan *Big Data* ditandai dengan warna jingga, dan alat untuk mengukur kinerja *big data* ditandai dengan warna hijau.

Tabel 3.2 Perangkat Lunak yang Dibutuhkan

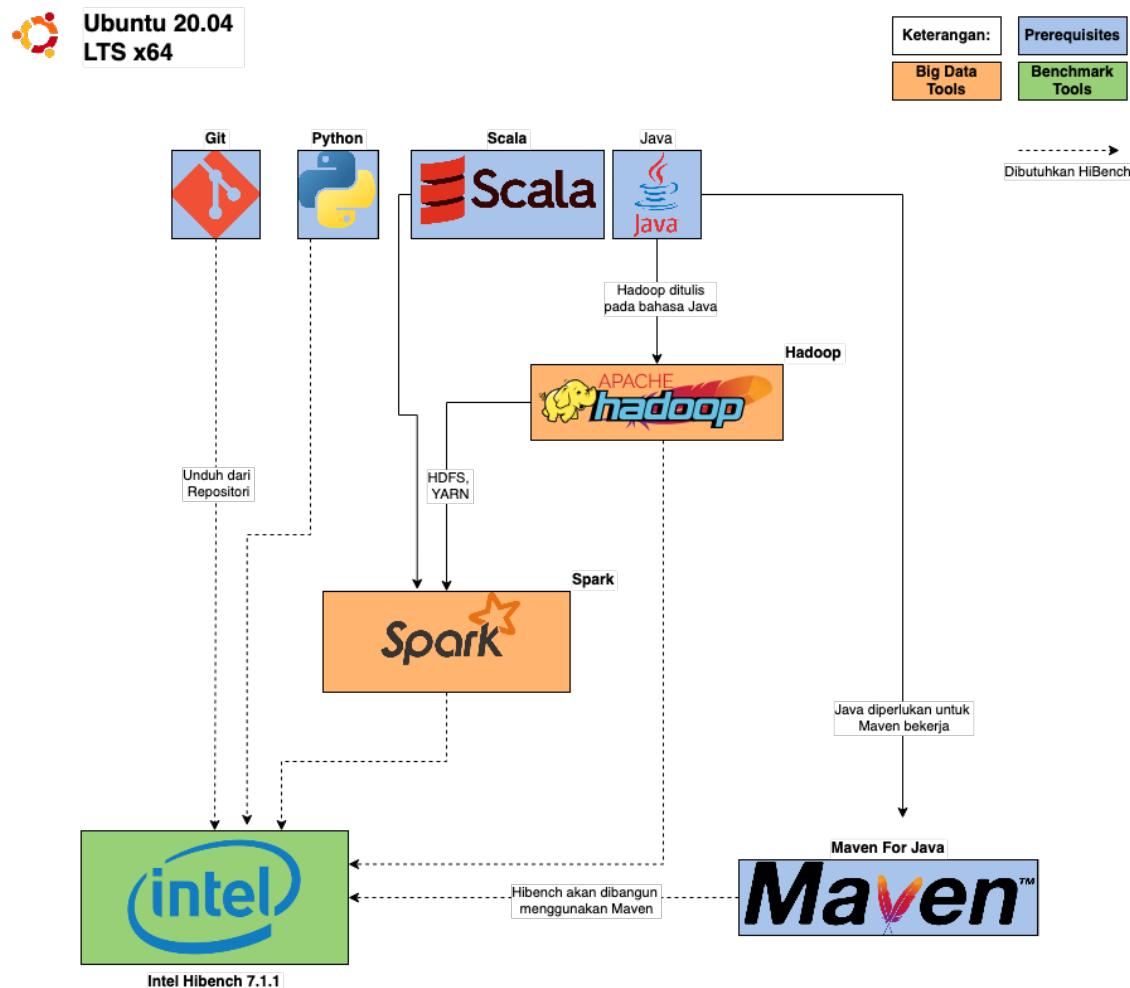
Perangkat Lunak	Deskripsi
Ubuntu 20.04 LTS x64	Sistem operasi Linux berbasis Ubuntu
Git	Sistem kontrol versi
Maven	Perangkat lunak manajemen proyek Java
Java 8	Bahasa pemrograman
Python 3.7	Bahasa pemrograman
Scala 2.11.8	Bahasa pemrograman
Hadoop 2.4	Perangkat lunak <i>big data</i>
Spark 2.1.3	Perangkat lunak <i>big data</i>
Hibench	Alat mengukur kinerja Hadoop dan Spark
Dool	Alat melihat penggunaan <i>resource</i> sistem

3.3.1 Instalasi Perangkat Lunak Prasyarat

Ada beberapa perangkat lunak yang perlu dipasang sebelum menggunakan Hadoop, Spark, dan HiBench, yaitu Ubuntu 20.04 LTS x64, Git, Java 8 dan Maven, Python 3.7, dan 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 3.2. Penjelasan lebih lengkap mengenai tata cara instalasi dan konfigurasi perangkat lunak prasyarat ini disajikan pada Lampiran B.



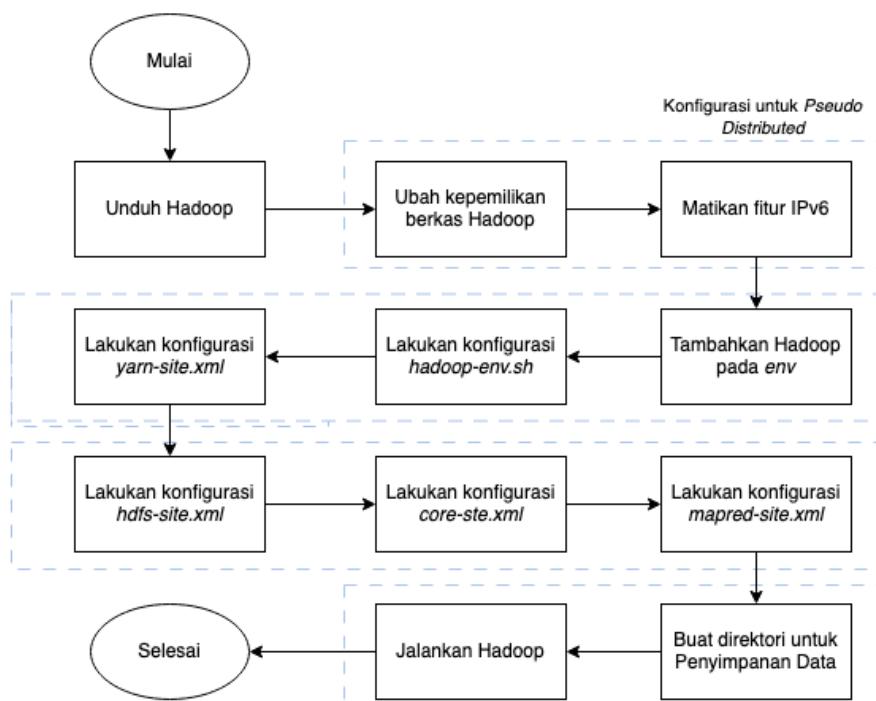
Gambar 3.2 Alur Instalasi Perangkat Lunak Prasyarat



Gambar 3.3 Alur Instalasi Perangkat Lunak

3.3.2 Instalasi dan Konfigurasi Hadoop

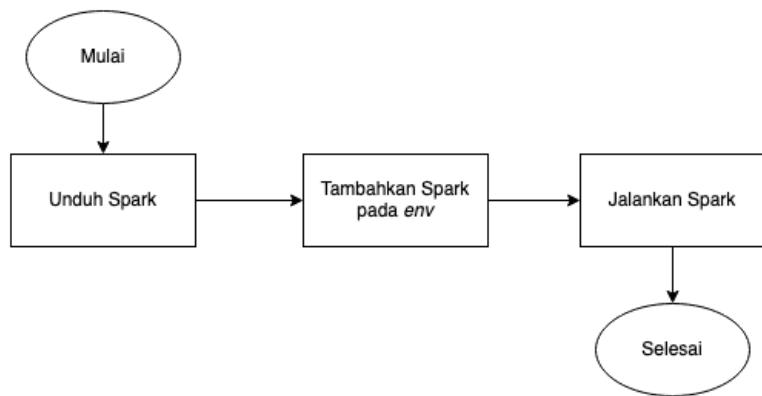
Hadoop adalah perangkat lunak *open source* yang efektif dalam menyimpan dan memproses data dalam skala besar. Hadoop memungkinkan pengklasteran beberapa komputer untuk menganalisis data besar secara paralel dengan lebih cepat. Setelah perangkat lunak prasyarat berhasil dipasang, Hadoop dapat dipasang mengikuti panduan lengkap pada Lampiran C. Konfigurasi Hadoop supaya dapat berjalan pada *pseudo distributed* meliputi pengubahan kepemilikan berkas Hadoop, mematikan fitur IPv6, menambahkan Hadoop pada *env*, dan mengatur konfigurasi komponen Hadoop sesuai dengan Gambar 3.4.



Gambar 3.4 Alur Instalasi dan Konfigurasi Hadoop

3.3.3 Instalasi dan Konfigurasi Spark

Spark dan Hadoop memiliki hubungan yang erat. Spark dapat berjalan di *Hadoop Distributed File System* (HDFS) dan dapat menggunakan Hadoop YARN sebagai manajer sumber daya. Oleh karena itu, instalasi Spark membutuhkan Hadoop 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.

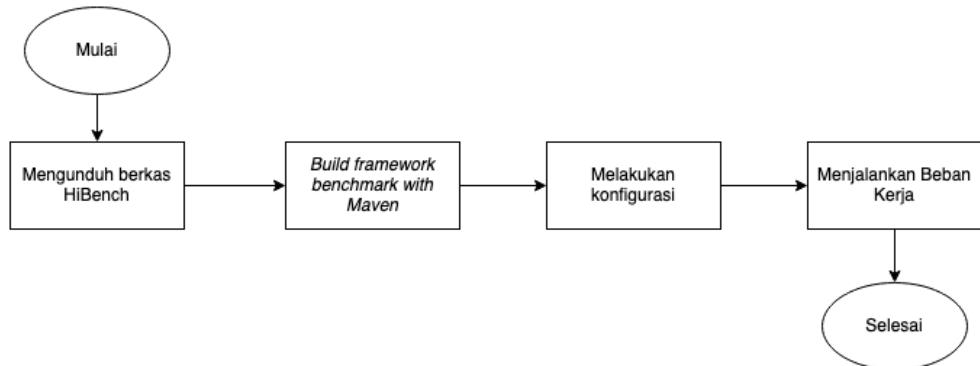


Gambar 3.5 Alur Instalasi dan Konfigurasi Spark

3.3.4 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.

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.



Gambar 3.6 Alur Instalasi dan Konfigurasi HiBench

3.4 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*.

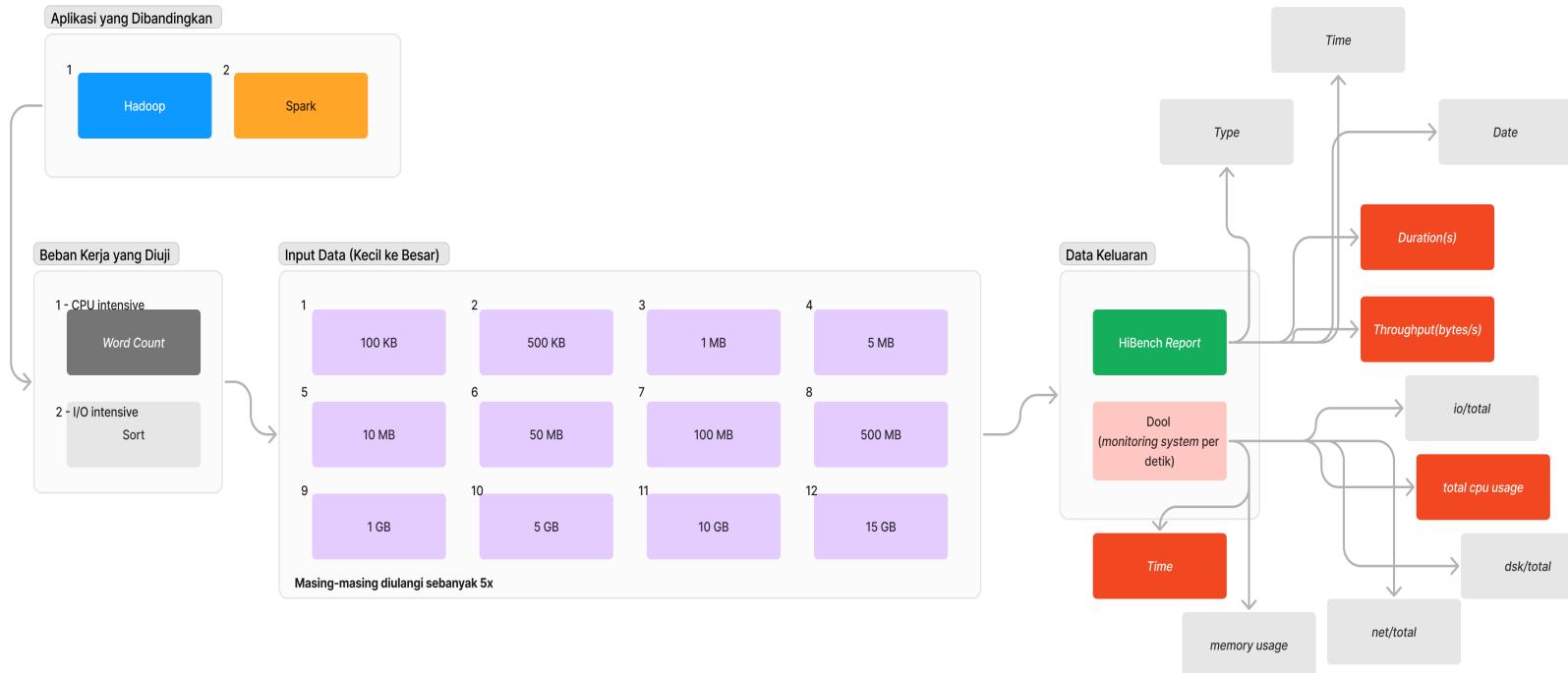
Penelitian ini difokuskan pada pengujian dua beban kerja yang umum dalam pemrosesan big data, yaitu *word count* dan *sort*. Beban kerja ini akan dieksekusi pada dua *platform big data* yang populer, yaitu Hadoop dan Spark. Setiap kombinasi *platform* dan beban kerja akan diuji dengan 12 ukuran data 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 lima 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. 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 (lima kali)

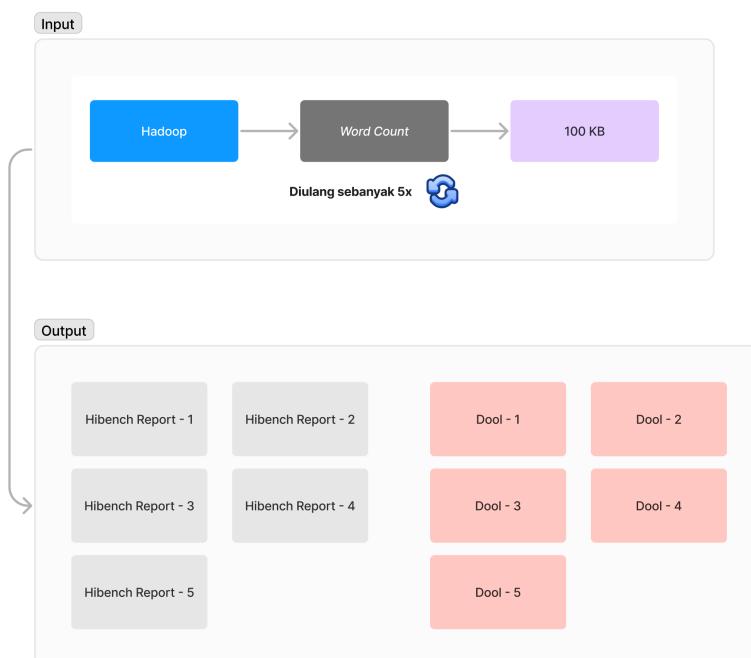


Gambar 3.7 End-to-end Penelitian

Tabel 3.3 Variasi Input Data

Label Input Data	Ukuran Input Data (bita)
100 KB	$100000 (1 \times 10^5)$
500 KB	$500000 (5 \times 10^5)$
1 MB	1×10^6
5 MB	5×10^6
10 MB	1×10^7
50 MB	5×10^7
100 MB	1×10^8
500 MB	5×10^8
1 GB	1×10^9
5 GB	5×10^9
10 GB	1×10^{10}
15 GB	1.5×10^{10}

Sebagai contoh, *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 3.8.



Gambar 3.8 Contoh Percobaan

Otomatisasi menjadi penting untuk memastikan efisiensi dan akurasi karena jumlah percobaan yang banyak. Skrip otomatisasi dikembangkan untuk mengotomatiskan seluruh proses eksperimen. 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 sebelum eksekusi beban kerja dilakukan. Setelah data siap, eksekusi beban kerja akan dijalankan. Setiap perulangan, perangkat lunak *Dool* dijalankan untuk memonitor aktivitas sistem. 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.

3.5 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.

2. Penggunaan Sumber Daya

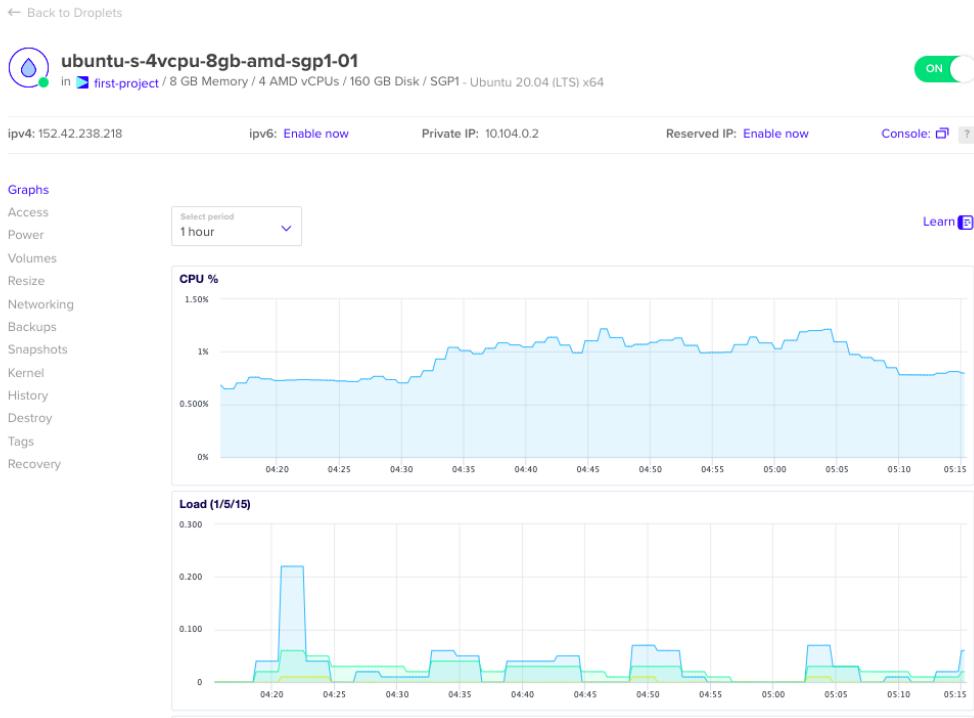
- (a) Penggunaan CPU. Bagian ini akan menganalisis penggunaan CPU oleh Hadoop dan Spark selama menjalankan berbagai beban kerja. Informasi ini dapat diperoleh dari berkas *monitoring* sistem yang dihasilkan oleh Dool. Analisis penggunaan CPU membantu memahami bagaimana setiap *platform* memanfaatkan 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 tanpa beban kerja.** 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 Tabel 4.2 merangkum konfigurasi parameter yang digunakan.

Tabel 4.1 Konfigurasi HiBench

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

Tabel 4.2 Konfigurasi Spark

Nama Parameter	Nilai Parameter	Keterangan Parameter
<i>hibench.yarn.executor.num</i>	2	Jumlah <i>executor</i>
<i>hibench.yarn.executor.cores</i>	4	Jumlah <i>core CPU</i> setiap <i>executor</i>
<i>spark.executor.memory</i>	4 GB	Jumlah memori setiap <i>executor</i>
<i>spark.driver.memory</i>	4 GB	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 4.7, terlihat bahwa ekstensi berkasnya *.report* dan terlihat beberapa data seperti jenis beban kerja, aplikasi yang digunakan, besar input data, durasi, dan *throughput*.

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	14489	14489
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	4056	4056
ScalaSparkWordcount	2024-04-17	09:47:30	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
ScalaSparkWordcount	2024-04-17	10:07:33	106383	27.200	3905	3905

Gambar 4.7 Data HiBench Report

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

dool 1.3.1 CSV output																									
Author:	Scott Baker						URL:		https://github.com/scottchiefbaker/dool/																
Host:	ubuntu-s-4vcpu-8gb-amd-sgp1-01						User:		hadoop																
Cmdline:	/home/hadoop/bin/dool --all --io --output sort-fivegig-1-hadoop.csv --bytes						Date:		17 Apr 2024 15:03:05 UTC																
total cpu usage							disk/total		net/total		paging		memory usage				system			procs		load avg			
usr	sys	idle	wal	stl	read	writ	recv	send	in	out	used	free	cach	aval	int	csw	run	bik	new	1m	5m	15m	r		
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	8.155	3.650	3.560	3.590	4	
45.297	3.218		51.485	0	0	1048576	40960		330	1356	0	0	2161164288	475455488	5445558272	5885804544	3089	4605	1	0	46	3.650	3.560	3.590	4
43.750	4		52	0	0.250	1048576	32768		264	1158	0	0	2165202944	470269952	5446606848	5881778176	2892	4249	1	0	30	3.650	3.560	3.590	4
60.759	10.127		24.051	5.063	0	6979584	286494720		198	966	0	0	2301308928	325865472	545432712	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	5965660160	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	9
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	C
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	C
93.734	6.266		0	0	0	376832	0		0	0	0	0	2678603776	1308016364	5476794888	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	5237037914	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	5134532606	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	4255154176	4292711363	5321	10527	16	0	10	3.800	3.590	3.600	2
87.500	12.500		0	0	0	301109248	0		0	0	0	0	3738468352	123019264	4271403008	4297218592	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	3739594752	120360960	476752384	4291170306	2814	3733	3	5	0	3.800	3.590	3.600	8
78.238	20.207		0.518	0.777	0.259	287088664	240549888		0	0	0	0	3513155584	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	3027361392	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	4155037494	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	1254203394	417019904	532052736	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	2
90.727	9.273		0	0	0	40775680	466944		0	0	0	0	3094884352	650993664	4389711872	4936105986	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	317709372	401768448	4555649492	4853927932	2614	8096	15	0	23	5.430	3.950	3.720	2
86	14		0	0	0	36384768	0		0	0	0	0	3560140800	253534208	4325302272	4471307952	2332	5864	9	0	39	5.430	3.950	3.720	2
87	12.750		0.250	0	0	34440832	32768		0	0	0	0	3503181824	132292608	4509560832	452746736	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	3266388568	336080896	4544860160	4764880896	2951	5181	7	0	50	5.430	3.950	3.720	2
81.612	14.610		2.771	1.008	0	8000168	297545728		66	290	0	0	3318931456	134598656	4691398656	4711890540	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

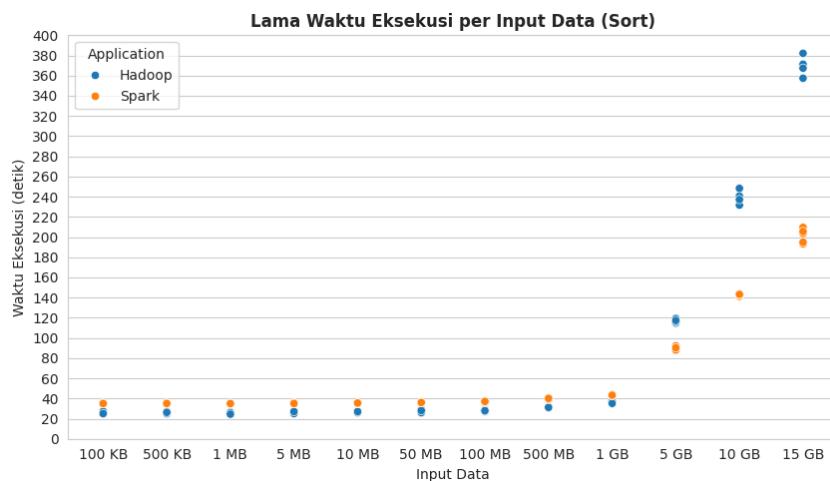
Gambar 4.8 Contoh Data Dool

4.5 Analisis dan Evaluasi Hasil Eksperimen: Kinerja

4.5.1 Persebaran Waktu Eksekusi pada Hadoop dan Spark

Gambar 4.9 dan Gambar 4.10 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.

Pada Gambar 4.9, 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 hingga 40 detik, sedangkan Spark berada pada rentang waktu 35 hingga 45 detik. Namun, untuk input data sebesar 5 GB, Spark menunjukkan waktu eksekusi yang lebih cepat dibandingkan Hadoop. Spark berada pada rentang 80 hingga 100 detik, sementara Hadoop berada pada rentang 110 hingga 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.



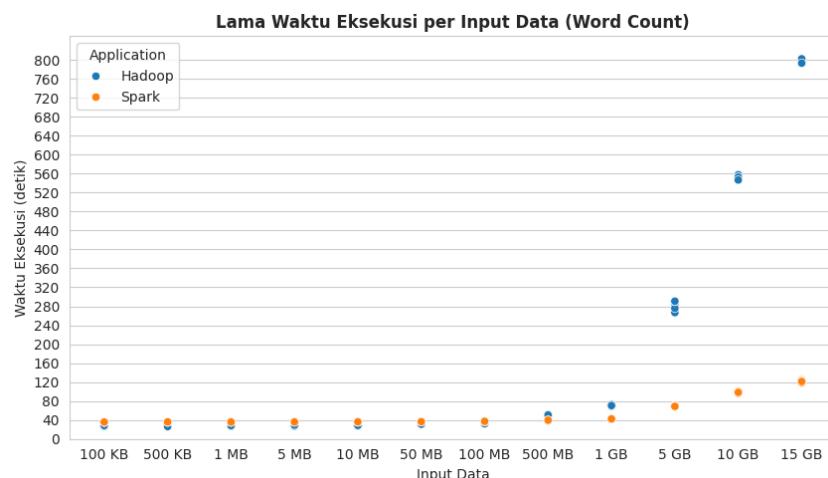
Gambar 4.9 Persebaran Waktu Eksekusi *Sort* (Hadoop, Spark)

Tabel 4.3 menunjukkan data yang lebih rinci mengenai durasi waktu eksekusi *sort* untuk Hadoop dan Spark. Berdasarkan tabel tersebut, terlihat bahwa untuk ukuran input data yang kecil hingga menengah, Hadoop relatif lebih cepat daripada Spark. Namun, seiring bertambahnya ukuran input data, Spark menunjukkan performa yang lebih baik dengan waktu eksekusi yang lebih singkat.

Tabel 4.3 Statistika Deskriptif Lama Waktu Eksekusi (*Sort*)

Aplikasi	Input Data	Waktu Eksekusi (maks)	Waktu Eksekusi (min)	Waktu Eksekusi (mean)	Standar Deviasi
Spark	100 KB	35.534	35.022	35.149	0.21661256
Hadoop	100 KB	27.275	25.033	26.139	1.06565449
Spark	500 KB	35.387	34.891	35.1468	0.19164603
Hadoop	500 KB	27.07	25.049	25.7052	0.91658508
Spark	1 MB	35.318	34.84	35.074	0.19876494
Hadoop	1 MB	26.619	24.439	25.9048	0.84623886
Spark	5 MB	35.992	35.057	35.35	0.38425317
Hadoop	5 MB	27.125	25.055	26.4528	0.89920087
Spark	10 MB	35.492	35.185	35.307	0.11954706
Hadoop	10 MB	27.02	25.96	26.4028	0.53875198
Spark	50 MB	36.284	35.716	36.0214	0.20281593
Hadoop	50 MB	29.059	25.942	27.2494	1.29796988
Spark	100 MB	37.099	36.137	36.614	0.35491196
Hadoop	100 MB	29.097	27.081	28.0776	0.72310601
Spark	500 MB	40.157	39.217	39.8564	0.42386826
Hadoop	500 MB	32.006	30.971	31.3254	0.39927597
Spark	1 GB	43.711	42.682	43.2808	0.38513855
Hadoop	1 GB	37.785	35.12	36.817	1.00034119
Spark	5 GB	92.115	87.748	89.6452	1.77748015
Hadoop	5 GB	119.26	114.71	116.6142	1.74711797
Spark	10 GB	143.217	141.211	142.5454	0.77448744
Hadoop	10 GB	248.255	231.66	239.2172	6.03409494
Spark	15 GB	209.59	193.234	201.3588	7.0346547
Hadoop	15 GB	382.111	357.422	369.7012	8.84744702

Pada Gambar 4.10, 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 hingga 40 detik, meskipun perbedaannya tidak signifikan dibandingkan Spark karena titik data Hadoop dan Spark saling berdekatan.



Gambar 4.10 Persebaran Waktu Eksekusi *Word Count* (Hadoop, Spark)

Tabel 4.4 menunjukkan data yang lebih rinci mengenai durasi waktu eksekusi *word count* untuk Hadoop dan Spark. Berdasarkan tabel tersebut, terlihat bahwa Hadoop lebih cepat daripada Spark untuk ukuran input data yang kecil hingga menengah yaitu pada 100 KB hingga 100 MB, tetapi Spark lebih cepat untuk ukuran input data yang besar.

Tabel 4.4 Statistika Deskriptif Lama Waktu Eksekusi (*Word Count*)

Aplikasi	Input Data	Waktu Eksekusi (maks)	Waktu Eksekusi (min)	Waktu Eksekusi (mean)	Standar Deviasi
Spark	100 KB	35.764	35.302	35.5502	0.1853
Hadoop	100 KB	28.186	26.364	27.2468	0.6466
Spark	500 KB	35.926	35.66	35.7744	0.1134
Hadoop	500 KB	27.272	26.235	26.8744	0.5328
Spark	1 MB	36.011	35.593	35.776	0.1627
Hadoop	1 MB	28.312	27.142	27.8554	0.574
Spark	5 MB	36.301	35.742	35.9918	0.2779
Hadoop	5 MB	29.242	27.241	28.2384	0.9601
Spark	10 MB	36.146	35.642	35.9442	0.2008
Hadoop	10 MB	29.161	27.361	28.2942	0.6543
Spark	50 MB	36.553	36.235	36.4104	0.1527
Hadoop	50 MB	31.228	31.078	31.165	0.0638
Spark	100 MB	37.097	36.634	36.9072	0.1708
Hadoop	100 MB	33.331	32.146	32.6726	0.5878
Spark	500 MB	39.596	38.722	39.1898	0.3405
Hadoop	500 MB	51.441	48.327	50.0346	1.1881
Spark	1 GB	42.343	42.052	42.164	0.1142
Hadoop	1 GB	72.507	68.653	70.3728	1.4179
Spark	5 GB	70.857	68.73	69.3636	0.8512
Hadoop	5 GB	290.775	266.665	281.3564	10.204
Spark	10 GB	100.229	96.36	98.5008	1.402
Hadoop	10 GB	557.603	546.448	551.6242	4.2258
Spark	15 GB	124.096	118.904	121.6664	1.8752
Hadoop	15 GB	802.291	793.122	797.8058	3.8834

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

- Untuk beban kerja *sort*, Spark lebih unggul mulai dari ukuran input data 5 GB, 10 GB, dan 15 GB.
- 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.

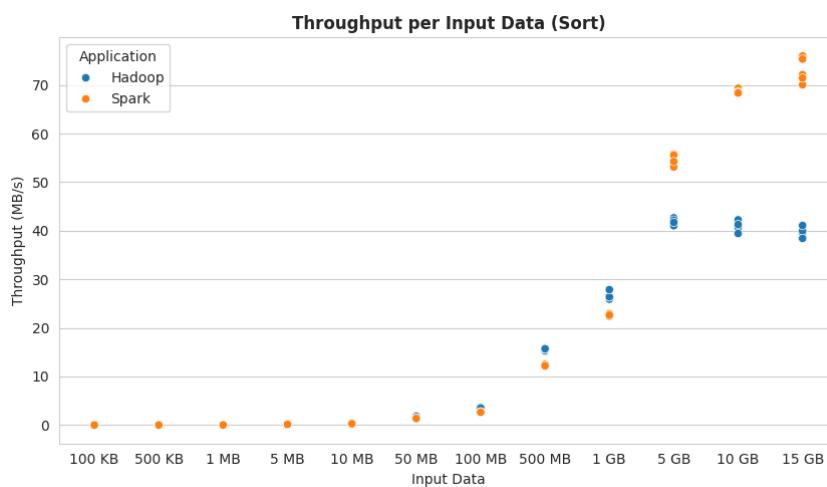
4.5.2 Persebaran *Throughput* pada Hadoop dan Spark

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 menyajikan *scatter plot* yang membandingkan *throughput* Hadoop dan Spark dalam dua tugas pemrosesan data, yaitu *sort* (Gambar 4.11) dan *word count* (Gambar 4.12). Sumbu x pada kedua gambar menunjukkan variasi ukuran input data, sedangkan sumbu y menunjukkan *throughput* dalam MB/s.

Pada tugas *sort* (Gambar 4.11), 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.



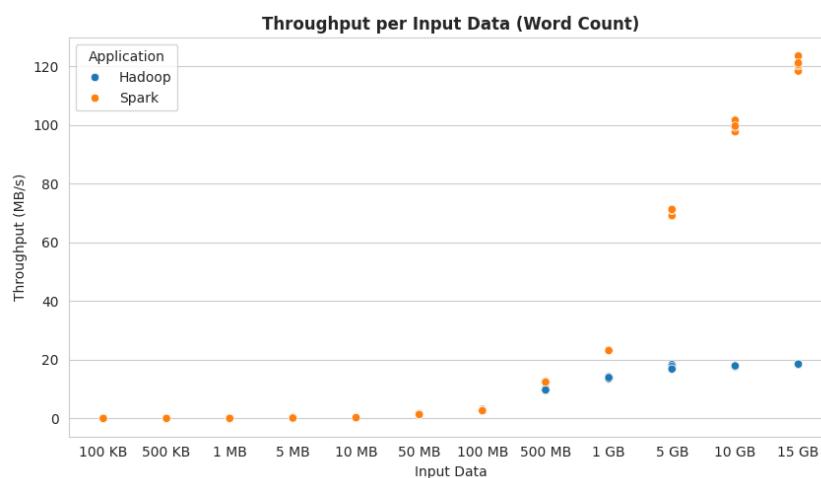
Gambar 4.11 Persebaran *Throughput Sort* (Hadoop, Spark)

Tabel 4.5 menunjukkan data yang lebih rinci mengenai *throughput sort* untuk Hadoop dan Spark. Berdasarkan tabel tersebut, terlihat bahwa Hadoop memiliki *throughput* yang lebih rendah dibandingkan Spark, terutama pada ukuran data yang lebih besar. Perbedaan kinerja ini dapat dilihat dari kolom "Throughput (maks)" dan "Throughput (min)". Misalnya, untuk input data 100 KB, Hadoop memiliki *throughput* maksimum 0.00403214 MB/s, sedangkan Spark memiliki *throughput* maksimum 0.00286674 MB/s. Ketika input data mencapai 10 GB, *throughput* maksimum Hadoop mencapai 42.2594395 MB/s, sementara Spark mencapai 69.3276262 MB/s.

Tabel 4.5 Statistika Deskriptif *Throughput (Sort)*

Aplikasi	Input Data	Throughput (maks)	Throughput (min)	Throughput (mean)	Standar Deviasi
Spark	100 KB	0.00286674	0.00282574	0.00285664	0.0000173
Hadoop	100 KB	0.00403214	0.00370121	0.00386696	0.0001574
Spark	500 KB	0.01411057	0.01391315	0.01400814	0.0000762
Hadoop	500 KB	0.01967716	0.01820755	0.01919365	0.0006699
Spark	1 MB	0.02818203	0.02780056	0.02799492	0.0001585
Hadoop	1 MB	0.04016113	0.03687286	0.03792267	0.0012865
Spark	5 MB	0.13967896	0.13605022	0.13853436	0.0014905
Hadoop	5 MB	0.19550419	0.18058491	0.18534927	0.0064644
Spark	10 MB	0.2783165	0.27590942	0.27735748	0.0009372
Hadoop	10 MB	0.3772049	0.36240768	0.37100182	0.0075152
Spark	50 MB	1.37057781	1.349123	1.35899258	0.0076652
Hadoop	50 MB	1.88702679	1.68461514	1.79971123	0.0846067
Spark	100 MB	2.70916271	2.6389122	2.67406902	0.0259209
Hadoop	100 MB	3.61521244	3.36473083	3.48874283	0.0898392
Spark	500 MB	12.4816198	12.1894493	12.282502	0.1315462
Hadoop	500 MB	15.8051376	15.2940359	15.6283312	0.1966307
Spark	1 GB	22.9367561	22.3968019	22.6208609	0.2022944
Hadoop	1 GB	27.8754454	25.9093733	26.6067383	0.7431688
Spark	5 GB	55.783987	53.1393728	54.6205183	1.0788883
Hadoop	5 GB	42.671937	41.0439196	41.9826252	0.6242206
Spark	10 GB	69.3276262	68.3565731	68.6802658	0.3753884
Hadoop	10 GB	42.2594395	39.4345398	40.9450817	1.0249329
Spark	15 GB	75.9945831	70.0641127	72.9998146	2.5623857
Hadoop	15 GB	41.0852346	38.4306307	39.7388464	0.9514776

Pada beban kerja *word count* (Gambar 4.12), Spark mencapai *throughput* yang lebih tinggi daripada Hadoop. Perbedaan *throughput* paling mencolok terlihat pada ukuran data terbesar (15 GB), dimana 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*.



Gambar 4.12 Persebaran *Throughput Word Count* (Hadoop, Spark)

Tabel 4.6 menunjukkan data yang lebih rinci mengenai *throughput word count* untuk Hadoop dan Spark. Berdasarkan tabel tersebut, terlihat bahwa Hadoop

memiliki *throughput* yang lebih rendah dibandingkan Spark, terutama pada ukuran data yang lebih besar. Perbedaan kinerja ini dapat dilihat dari kolom "Throughput (maks)" dan "Throughput (min)". Misalnya, untuk input data 100 KB, Hadoop memiliki *throughput* maksimum 0.00384808 MB/s, sedangkan Spark memiliki *throughput* maksimum 0.00286102 MB/s. Ketika input data mencapai 10 GB, *throughput* maksimum Hadoop mencapai 17.9153442 MB/s, sementara Spark mencapai 101.596455 MB/s.

Tabel 4.6 Statistika Deskriptif *Throughput* (*Word Count*)

Aplikasi	Input Data	Throughput (maks)	Throughput (min)	Throughput (mean)	Standar Deviasi
Spark	100 KB	0.00286102	0.00282383	0.002841	0.0000148
Hadoop	100 KB	0.00384808	0.00359917	0.00372486	0.0000882
Spark	500 KB	0.01383686	0.01373386	0.01379242	0.0000438
Hadoop	500 KB	0.01875591	0.01804256	0.01831551	0.0003658
Spark	1 MB	0.02759743	0.02727699	0.02745667	0.0001247
Hadoop	1 MB	0.0361557	0.03466225	0.03524208	0.000732
Spark	5 MB	0.13701153	0.134902	0.1360672	0.0010482
Hadoop	5 MB	0.17980003	0.16749668	0.17360954	0.0058914
Spark	10 MB	0.27474499	0.27091408	0.27244186	0.001526
Hadoop	10 MB	0.3579216	0.33582878	0.34626541	0.0080376
Spark	50 MB	1.351017	1.33926392	1.34452782	0.0056451
Hadoop	50 MB	1.57513809	1.56757259	1.57074642	0.0032184
Spark	100 MB	2.67248154	2.63912678	2.6527441	0.0123136
Hadoop	100 MB	3.04554081	2.93726444	2.99722595	0.0536088
Spark	500 MB	12.6410389	12.3620138	12.490901	0.1087093
Hadoop	500 MB	10.128686	9.51554203	9.78746166	0.2343767
Spark	1 GB	23.2802544	23.1202621	23.2185509	0.0627568
Hadoop	1 GB	14.259717	13.5017633	13.9157192	0.2784217
Spark	5 GB	71.219182	69.0813093	70.5770096	0.8534114
Hadoop	5 GB	18.3560066	16.8339939	17.4161545	0.6423301
Spark	10 GB	101.596455	97.6746683	99.4045837	1.4236794
Hadoop	10 GB	17.9153442	17.5569429	17.7480667	0.135773
Spark	15 GB	123.5007	118.333607	120.719687	1.8684413
Hadoop	15 GB	18.5150871	18.3034868	18.4067369	0.0896672

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 berukuran besar, dengan rincian sebagai berikut:

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

4.6 Analisis dan Evaluasi Hasil Eksperimen: Penggunaan Sumber Daya

4.6.1 Penggunaan CPU

Gambar 4.13 dan Gambar 4.14 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 pada gambar menandakan titik perpotongan Hadoop dan Spark pada waktu yang sama. Titik hitam tersebut berfungsi sebagai batas kondisi dimana penggunaan CPU Hadoop lebih tinggi dibandingkan Spark atau sebaliknya. Selanjutnya, titik hitam ini akan memberikan informasi visual mengenai perubahan dominasi penggunaan CPU antara Hadoop dan Spark selama proses komputasi.

Tabel 4.7 dan Tabel 4.8 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 beban kerja *sort* seperti yang terlihat pada Gambar 4.13, perbedaan pola penggunaan CPU antara Hadoop dan Spark terlihat pada ukuran data yang lebih besar. Jika dilihat pada input data 100 KB hingga 1 GB, penggunaan CPU pada masing-masing Hadoop dan Spark tidak terlihat perbedaannya. Selanjutnya, pada input data 5 GB, 10 GB, dan 15 GB, penggunaan CPU Hadoop sangat fluktuatif berkisar pada 60% sampai 95% pada dua per tiga bagian waktu awal eksekusi, dan satu per tiga waktu akhir eksekusi mengalami penurunan yang berkisar pada 20% sampai 80%. Berbeda dengan Hadoop, Spark memiliki penggunaan CPU yang lebih stabil, yaitu terkadang pada 60% sampai 80%, dan terkadang juga pada 80% sampai 100%).

Pada Tabel 4.7, terlihat bahwa durasi $U_h > U_s$ secara konsisten lebih tinggi dibandingkan dengan $U_h < U_s$ pada semua ukuran data. Hal ini menunjukkan bahwa Hadoop menggunakan lebih banyak CPU dibandingkan Spark untuk beban kerja *sort*. 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 beban kerja *word count* seperti yang terlihat pada Gambar 4.14, Hadoop menunjukkan penggunaan CPU yang lebih fluktuatif (naik turun) dan tinggi secara keseluruhan dibandingkan dengan Spark. Penggunaan CPU pada waktu ke-0

Tabel 4.7 Perbandingan *State (Sort)*

Input Data	Kondisi (s)	
	Uh < Us	Uh > Us
100 KB	15.2955246	21.7044754
500 KB	15.7152478	21.2847522
1 MB	15.1296715	21.8703285
5 MB	14.8516413	22.1483587
10 MB	14.0988502	22.9011498
50 MB	13.7726706	24.2273294
100 MB	14.2138896	23.7861104
500 MB	14.3734577	26.6265423
1 GB	16.2881796	28.7118204
5 GB	24.1825795	92.8174205
10 GB	32.6343114	207.365689
15 GB	57.1364482	314.863552

hingga ke-10 pada Hadoop berkisar pada 10% sampai 60%. Setelah itu, penggunaan CPU Hadoop naik sampai ke 100% dan naik turun. Jika dilihat dari input data yang lebih besar yaitu 15 GB, Hadoop memiliki pola CPU yang hampir sama pada setiap data input, yaitu

1. Sepuluh detik pertama penggunaan CPU berada pada 10% sampai 60%
2. Setengah waktu eksekusi naik turun pada penggunaan CPU 60% hingga 100%
3. Setengah waktu eksekusi terakhir menurunkan penggunaan CPU pada 50% sampai 90%

Selanjutnya, Spark juga memiliki pola tersendiri. Penggunaan CPU Spark pada detik pertama sampai detik ke lima belas fluktuatif pada 20% hingga 50%. Pada waktu ke-16 sampai waktu ke-35 turun hingga 0%, dan naik kembali pada waktu ke-35. Hal yang menarik lainnya adalah penggunaan CPU Spark tidak menyentuh 100%, tetapi memiliki waktu eksekusi beban kerja yang lebih cepat pada *word count*.

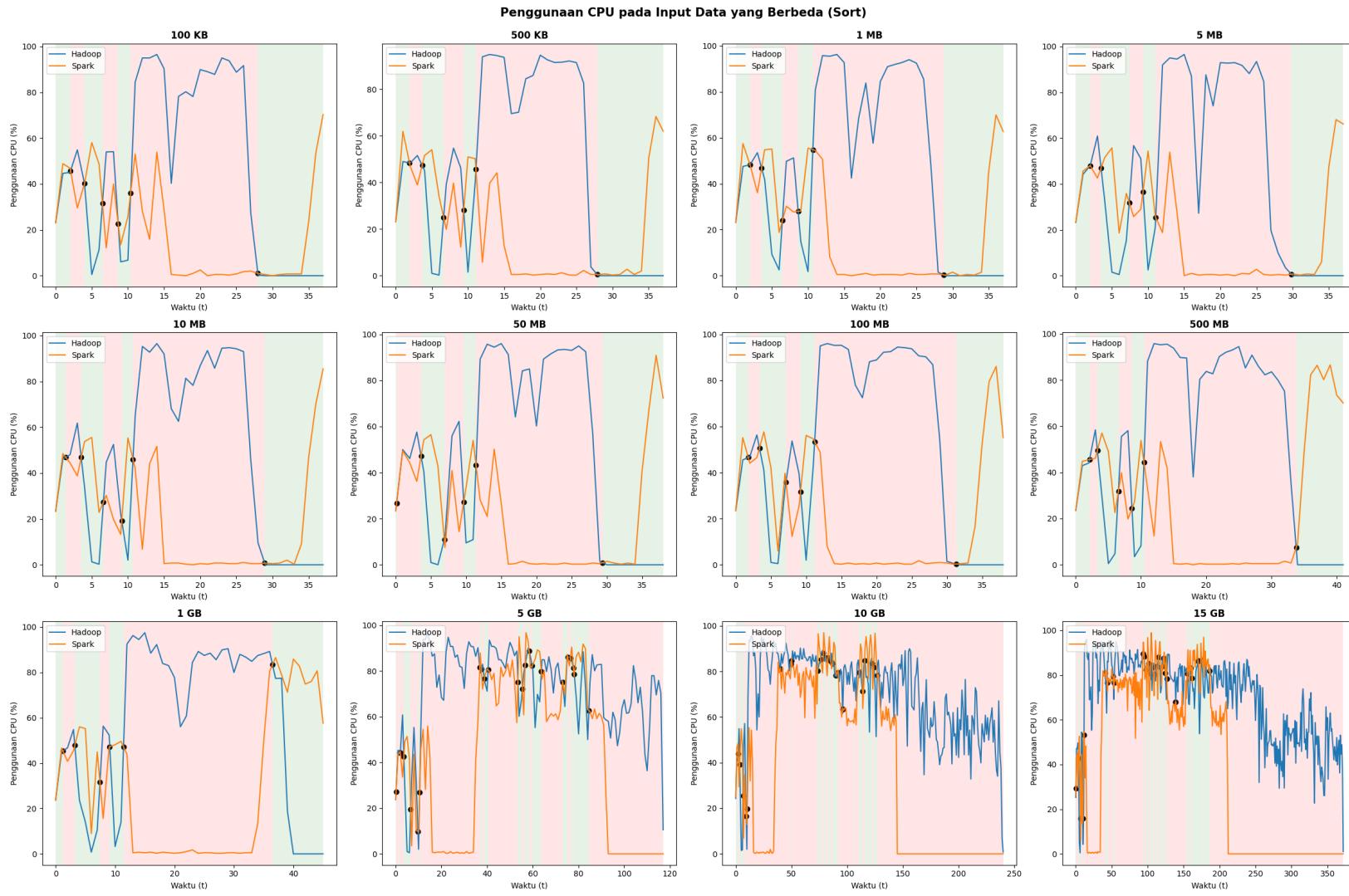
Pada Tabel 4.8, variasi dalam perbandingan *state word count* lebih terlihat dibandingkan *sort*. Secara umum, durasi $U_h > U_s$ tetap lebih tinggi dibandingkan $U_h < U_s$. Pada beban kerja *word count*, durasi $U_h < U_s$ tidak menunjukkan pola yang konsisten dengan ukuran data, tetapi durasi $U_h > U_s$ 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.

Tabel 4.8 Perbandingan *State* (*Word Count*)

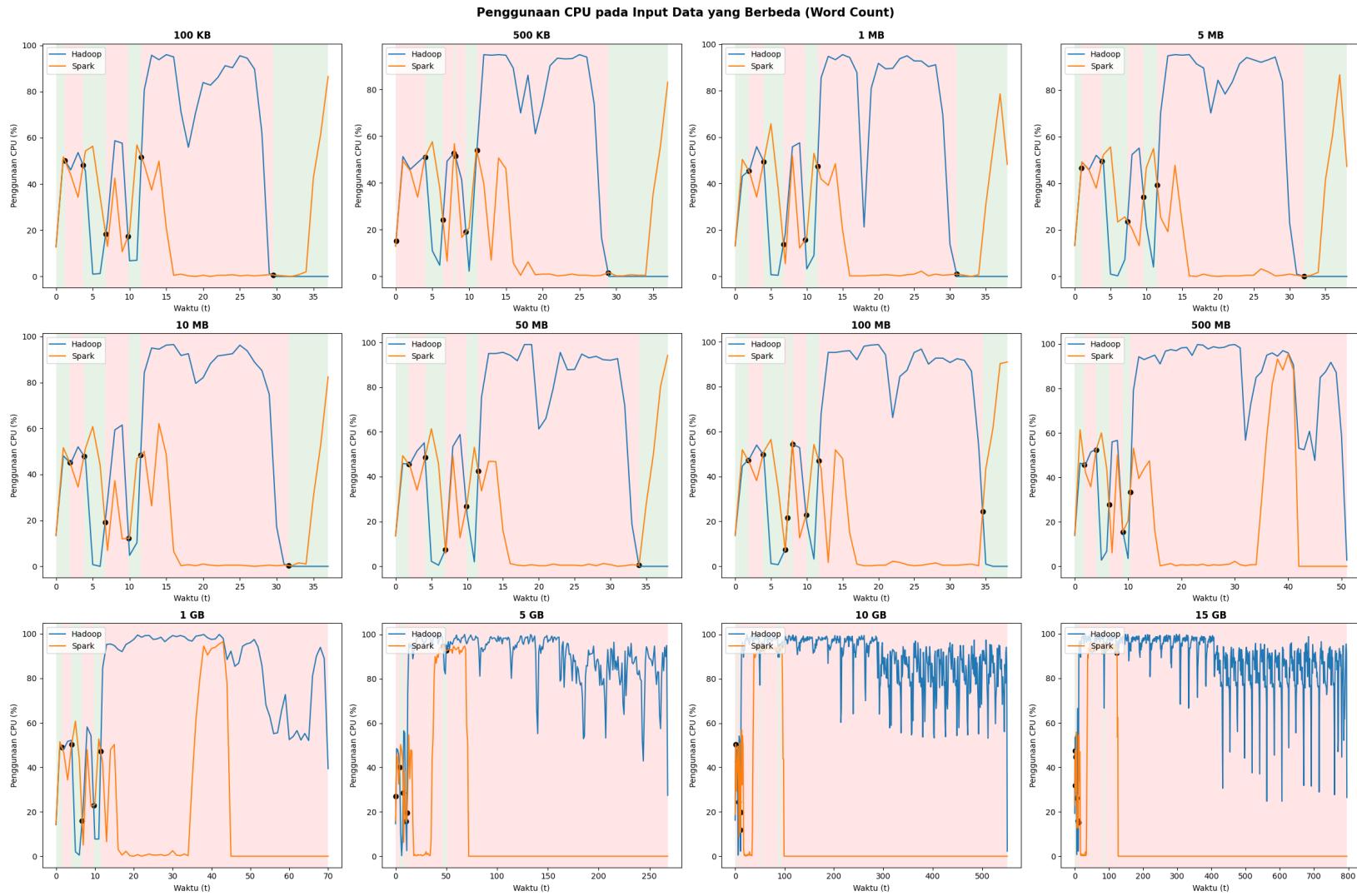
Input Data	Kondisi (s)	
	Uh < Us	Uh > Us
100 KB	13.5756308	23.4243692
500 KB	12.2835537	24.7164463
1 MB	13.4686789	24.5313211
5 MB	12.3929733	25.6070267
10 MB	11.7220593	25.2779407
50 MB	10.3495898	27.6504102
100 MB	10.8210338	27.1789662
500 MB	5.60362892	45.3963711
1 GB	6.03855036	63.9614496
5 GB	7.81560756	260.184392
10 GB	15.1215645	535.878435
15 GB	12.9367213	783.063279

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 4.13 Penggunaan CPU (Sort)



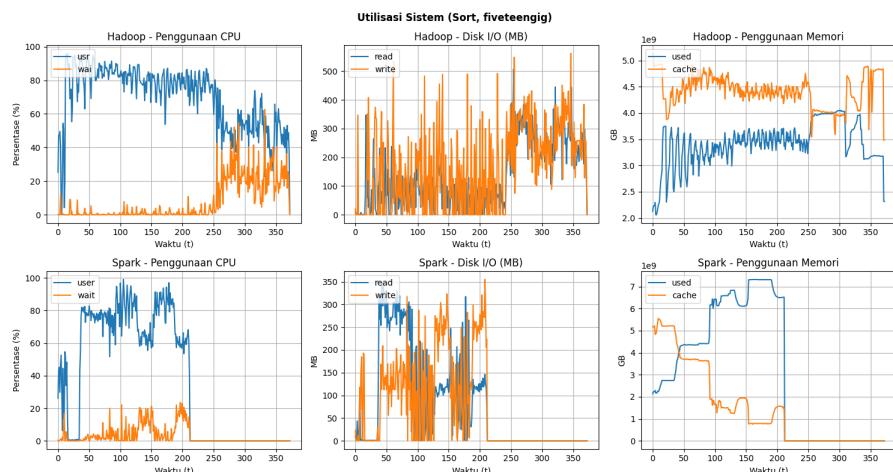
Gambar 4.14 Penggunaan CPU (Word Count)

4.6.2 Utilisasi Sistem

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, Penggunaan CPU (%), Disk I/O (MB), 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 terbesar (15 GB).

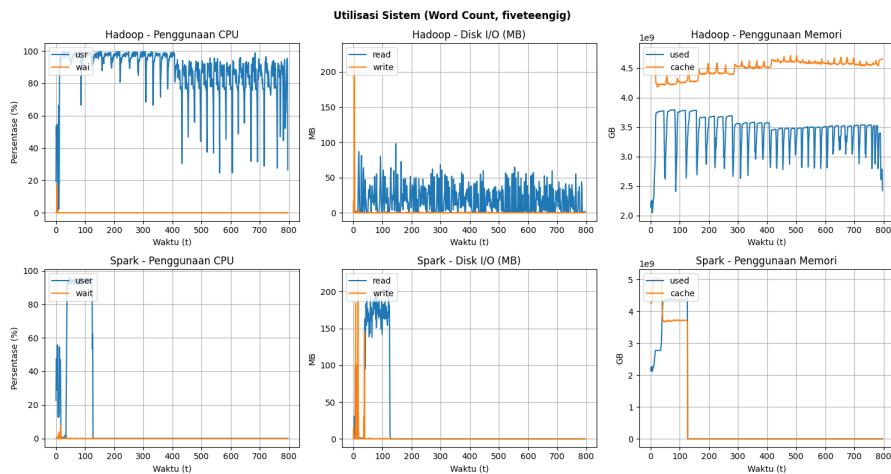
Pada beban kerja *sort* dan input data terbesar, yaitu 15 GB (seperti yang ditunjukkan pada Gambar 4.15), utilisasi sistem pada Hadoop dan Spark terlihat jelas perbedaannya. Jika dilihat pada penggunaan CPU, Hadoop membutuhkan waktu penggunaan CPU yang lebih lama, yaitu sekitar 370 detik, dimana Spark 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" 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 hingga 7 GB, dimana Hadoop hanya membutuhkan memori sekitar 3.5 GB.



Gambar 4.15 Utilisasi Sistem (*Sort*) pada Input Data 15 GB

Pada beban kerja *word count* dan input data terbesar, yaitu 15 GB (seperti yang ditunjukkan pada Gambar 4.16), utilisasi sistem pada Hadoop dan Spark 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 dilakukan oleh Hadoop berada pada ukuran 50 hingga 100 MB setiap waktunya. Pada Spark, aktivitas baca tersebut berukuran 150 hingga 200 MB. Kemudian, jika ditinjau melalui penggunaan memori, memori yang dibutuhkan Hadoop berkisar pada 2.5 hingga 3.7 GB. Pada Spark, memori yang dibutuhkan berkisar pada 2 hingga 4.5 GB.



Gambar 4.16 Utilisasi Sistem (*Word Count*) pada Input Data 15 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.

4.7 Perbandingan dengan Penelitian Sebelumnya [1]

Penelitian terdahulu yang telah dilakukan oleh Yassir Samadi, dkk. mendapatkan hasil seperti pada Tabel 4.9. Pada gambar tersebut terlihat bahwa pada input data 1 GB dan pada beban kerja *word count*, terjadi peningkatan performa sebesar 2.92, pada input data 5 GB sebesar 3.64, dan begitu juga pada input data 10 GB sebesar 3.52.

Tabel 4.9 Rasio Peningkatan Performa Spark-Hadoop [1]

Aplikasi	Input Data	Rata-rata Waktu Eksekusi (s)	Rasio Peningkatan
Spark	1 GB	47,852	
Hadoop	1 GB	139,712	2,92
Spark	5 GB	162,585	
Hadoop	5 GB	592,503	3,64
Spark	10 GB	292,665	
Hadoop	10 GB	1029,36	3,52

Penelitian ini menghasilkan hasil yang serupa, seperti yang ditunjukkan pada Tabel 4.10. 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 4.10 Rasio Peningkatan Performa Spark-Hadoop

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

Pada Tabel 4.10, 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.

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 beban kerja *machine learning*, dan menganalisis pengaruh penambahan jumlah *node* terhadap skalabilitas. 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.

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] 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).
- [9] 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.
- [10] *Hadoop Market Size, Share & Competition Analysis, 2021-2027*, <https://www.kbvresearch.com/hadoop-market/>. (dikunjungi pd. 05/15/2024).
- [11] 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).
- [12] 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).
 - [13] 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).
 - [14] S. Huang, J. Huang, J. Dai, T. Xie, dan B. Huang, “The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis”,
 - [15] C. Wibawa, S. Wirawan, M. Mustikasari, dan D. Anggraeni, “KOMPARASI KECEPATAN HADOOP MAPREDUCE DAN APACHE SPARK DALAM MENGOLAH DATA TEKS”, *Jurnal Ilmiah Matrik*, vol. 24, hlmn. 10–20, Apr. 2022.
 - [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 HiBench 7.0*. 2018. (dikunjungi pd. 12/28/2023).
 - [19] A. Oussous, F.-Z. Benjelloun, A. Ait Lahcen, dan S. Belfkih, “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] S. M. Ross, *Introductory Statistics*, Fourth edition. Amsterdam Boston Heidelberg London New York Oxford Paris San Diego San Francisco Singapore Sydney Tokyo: Elsevier Academic Press, 2017.
- [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] baeldung, *Guide to the “Cpu-Bound” and “I/O Bound” Terms | Baeldung on Computer Science*, <https://www.baeldung.com/cs/cpu-io-bound>, Des. 2021. (dikunjungi pd. 05/21/2024).
- [28] 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).
- [29] 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).
- [30] H. Herodotou, *Hadoop Performance Models*, Juni 2011. arXiv: 1106.0940 [cs]. (dikunjungi pd. 11/08/2023).
- [31] 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).
- [32] 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).
- [33] Apache Hadoop, <https://hadoop.apache.org/>. (dikunjungi pd. 11/08/2023).
- [34] 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).

- [35] 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).
- [36] T. John dan P. Misra, *Data Lake for Enterprises: Leveraging Lambda Architecture for Building Enterprise Data Lake*. Birmingham, UK, Mumbai: Packt Publishing, 2017.
- [37] 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).
- [38] 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).
- [39] *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).
- [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. (dikunjungi pd. 12/30/2023).
- [43] *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).
- [44] *Penjelasan Apa itu Bandwidth, Throughput, dan Latency*, <https://www.initialboard.com/penjelasan-apa-itu-bandwidth-throughput-dan-latency>, Jan. 2023. (dikunjungi pd. 05/21/2024).

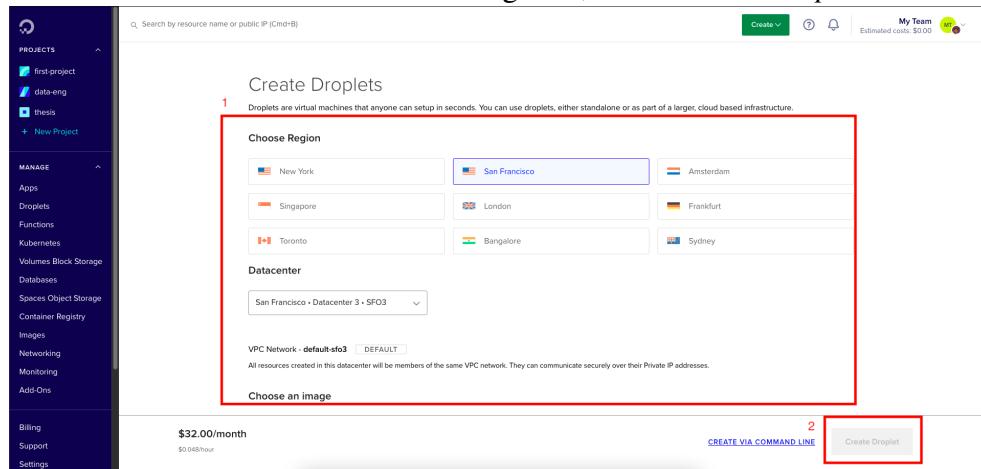
LAMPIRAN

LAMPIRAN A

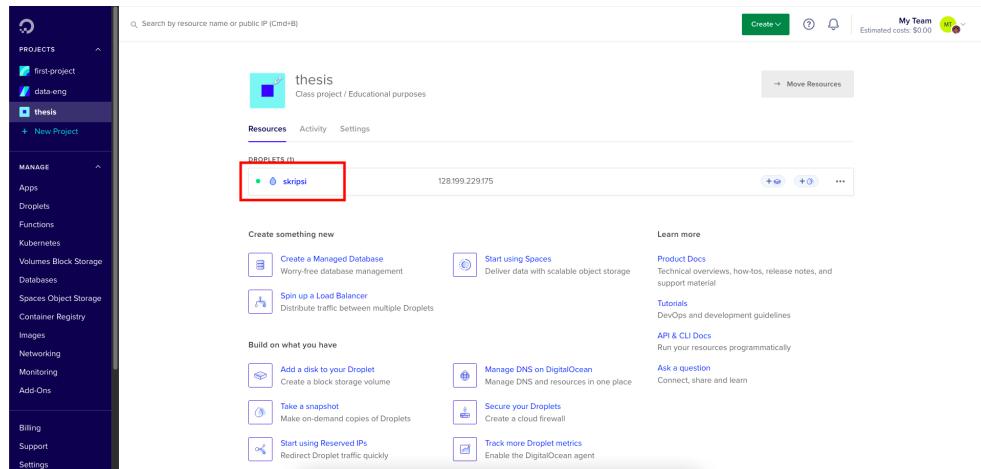
Pembuatan *Virtual Machine* (VM) pada DigitalOcean

Langkah-langkah pembuatan VM pada DigitalOcean dijelaskan seperti berikut,

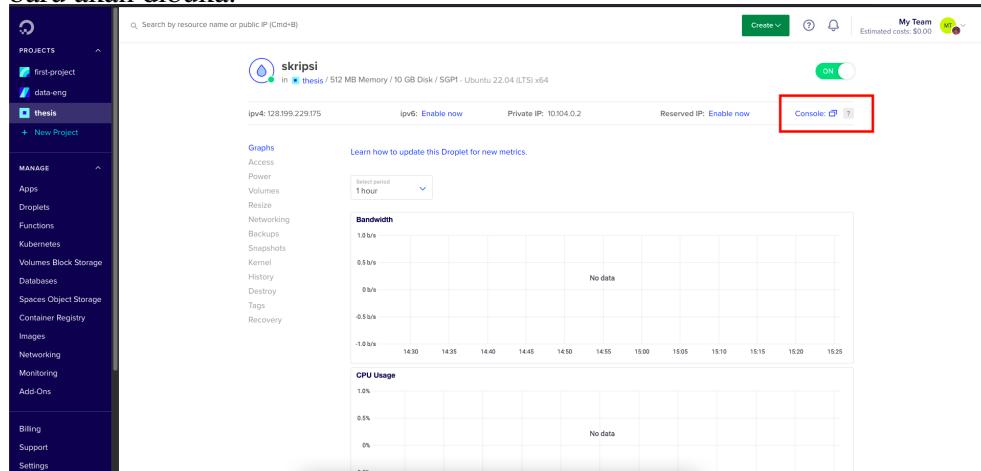
1. Buatlah akun DigitalOcean terlebih dahulu. Jika belum memiliki akun DigitalOcean, disarankan untuk mendaftar melalui *GitHub Student Developer Pack* sehingga nantinya akan diberikan kredit \$200 secara gratis. Jika sudah memiliki akun DigitalOcean, silakan melakukan *login*.
2. Perhatikan menu di 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.
3. Pada laman pembuatan Droplets, lakukan konfigurasi sesuai dengan Tabel 3.1. Jika telah selesai melakukan konfigurasi, tekan *Create Droplets*.



4. 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.



5. 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.



LAMPIRAN B

Instalasi dan Konfigurasi 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. Jika Droplets baru saja dibuat, perlu dilakukan pembaruan *index* pada *package management*. Pembaruan *package management* dapat dilakukan dengan `sudo apt update`.
2. 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`.
3. 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`.

4. 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`.

5. 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`.

6. 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`.

7. 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=$HADOOP_HOME/lib/native
13 export HADOOP_OPTS=-Djava.library.path=$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/\
6 hdfs/namenode
7 sudo chown -R hdfsuser:hadoop /usr/local/hadoop/yarn_data/\
8 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 -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}/bin/hadoop
6
7 # Hadoop configraution directory
8 hibench.hadoop.configure.dir ${hibench.hadoop.home}/etc/hadoop
9
10 # The root HDFS path to store HiBench data
11 hibench.hdfs.master      hdfs://localhost:9000
12
13 # Hadoop release provider. Supported value: apache
14 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-bebankerja>/prepare/prepare.sh
2 bin/workloads/micro/<nama-bebankerja>/hadoop/run.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     "limapuluhmb"
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 \
38             $scale/" conf/hibench.conf
39
40         # Tahap persiapan Hadoop
41         while true; do
42             bin/workloads/micro/$workload/prepare/prepare.sh
43             if [[ $? -eq 0 ]]; then
44                 break
45             fi
46             echo "      Tahap persiapan Hadoop gagal. Mencoba lagi..."
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/null 2>&1 &
56             dool_pid=$! # Menyimpan PID proses dool
57
58             # Menjalankan benchmark Hadoop
59             bin/workloads/micro/$workload/hadoop/run.sh
60
61             # Menghentikan dool setelah benchmark selesai
62             kill $dool_pid
63             wait $dool_pid 2>/dev/null
64         done
65
66         # Tahap persiapan Spark
```

```

67     while true; do
68         bin/workloads/micro/$workload/prepare/prepare.sh
69         if [[ $? -eq 0 ]]; then
70             break
71         fi
72         echo "    Tahap persiapan Spark gagal. Mencoba lagi..."
73     done
74
75     # Looping untuk setiap pengulangan Spark
76     for ((i = 1; i <= repetitions; i++)); do
77         echo "    Percobaan Spark $i"
78
79         # Mulai dool di latar belakang
80         nohup /home/hadoop/bin/dool --all --io --output \
81         "$workload-$scale-$i-spark.csv" --bytes > /dev/null 2>&1 &
82         dool_pid=$!
83
84         # Menjalankan benchmark Spark
85         bin/workloads/micro/$workload/spark/run.sh
86
87         # Menghentikan dool setelah benchmark selesai
88         kill $dool_pid
89         wait $dool_pid 2>/dev/null
90     done
91
92     # Tunggu 15 detik sebelum beralih ke skala berikutnya
93     sleep 15
94     done
95 done
96
97 echo "Selesai."

```
