



**北京理工大学**  
BEIJING INSTITUTE OF TECHNOLOGY

# 大数据系统开发——搜索引擎实现 实验报告

学    院： 计算机学院

专    业： 计算机科学与技术

任课教师： 冯恺宇

小组成员： 小组成员如下

教学班级	姓名	学号
07112303	左逸龙	1120231863
07112304	胡艺镭	1120232848
07112303	李雨桐	1120231975
07112304	周鑫	1120232701
07112303	黄奕晨	1120232530
07112303	刘兆钰	1120231933

二〇二五年九月二十九日

## 目录

<b>1. 实验概述</b>	<b>1</b>
1.1 实验目的	1
1.2 实验要求	1
1.3 实验环境	2
<b>2. 实验环境搭建</b>	<b>2</b>
2.1 虚拟机与网络配置	2
2.2 Hadoop 集群配置	5
2.3 HBase 集群配置	8
2.4 集群启动与验证	11
<b>3. 实验过程与实现</b>	<b>15</b>
3.1 数据准备	15
3.2 核心算法与代码实现	18
3.2.1 Mapper 实现	18
3.2.2 Reducer 实现	19
3.2.3 Driver 实现	21
3.3 程序打包与执行	23
<b>4. 实验结果与分析</b>	<b>24</b>
4.1 运行过程监控	24
4.2 结果验证	24
4.3 性能分析	25
<b>5. 遇到的问题及解决方案</b>	<b>26</b>
<b>6. 总结与心得</b>	<b>27</b>
6.1 实验总结	27
6.2 心得体会	28

## 1. 实验概述

### 1.1 实验目的

本实验旨在通过实际的系统部署与编程实现，使我们能够在大数据处理的完整流程中掌握核心技术与方法。具体目标如下：

1. 掌握 Hadoop、Zookeeper 及 HBase 完全分布式集群的搭建与配置方法，理解其在大数据系统中的角色与协作关系。
2. 深刻理解 MapReduce 编程模型，能够独立编写 Mapper 与 Reducer 程序，对大规模文本数据进行倒排索引的构建与统计。
3. 学习如何将 MapReduce 的处理结果与 HBase 数据库进行无缝集成，提升数据的存储与检索效率。
4. 系统体验从数据准备、分布式计算到结果存储的完整大数据处理流程，增强对大数据平台实际应用场景的理解与动手能力。

### 1.2 实验要求

根据课程实验的统一要求，本次实验的核心任务是基于 Hadoop 分布式计算平台完成倒排索引的构建与存储。具体包括以下内容：

1. 对提供的数据集 `sentences.txt` 进行预处理，将原始大规模文本按照行号和内容切分，并划分为若干子文件以便于分布式处理。
2. 编写 MapReduce 程序，实现倒排索引的构建过程：在 Map 阶段对文本进行分词并生成中间键值对，在 Reduce 阶段聚合结果并统计单词在不同文档中的出现频次。
3. 将 MapReduce 的最终处理结果写入 HBase 数据库中，形成便于快速检索的索引结构。
4. 验证索引结果的正确性与完整性，确保能够基于关键词实现高效的检索操作。

本实验不仅要求我们正确完成倒排索引的分布式实现与数据库存储，还强调对 Hadoop 生态系统各组件（HDFS、YARN、MapReduce、HBase）的综合应用与理解。

### 1.3 实验环境

本实验在虚拟化环境下完成，采用 CentOS 作为操作系统，并基于 Hadoop 完全分布式集群架构进行搭建。在软件环境方面，实验使用 JDK 1.8 作为运行时支撑，Hadoop 作为大数据分布式存储与计算框架，HBase 作为 NoSQL 数据存储系统，Zookeeper 用于提供分布式协调服务。此外，实验过程中借助 IntelliJ IDEA 与 Maven 进行 MapReduce 程序的开发与依赖管理。整体环境配置如表 1 所示。

表 1 实验软硬件环境

组件	版本/规格	备注
操作系统	CentOS 7.7.1908 (Core)	运行于 VMware 虚拟机
虚拟化软件	VMware Workstation 16 Pro	
JDK	1.8.0_241	
Hadoop	3.3.0	
HBase	2.1.0	
Zookeeper	3.7.1	
开发工具	IntelliJ IDEA, Maven	用于 Java 代码编写与打包

通过上述配置，实验能够在模拟的分布式环境中完整运行，从而为倒排索引的实现与验证提供可靠的基础支撑。

## 2. 实验环境搭建

### 2.1 虚拟机与网络配置

在完成 Hadoop、ZooKeeper 与 HBase 的安装前，需要配置好虚拟机的网络环境，保证多节点之间的通信与互信。

**1. 克隆虚拟机** 首先在 VMware 中安装并配置好一台基础 CentOS 系统，完成 JDK 等基础环境安装后，将该虚拟机作为母机进行克隆。克隆得到的多台虚拟机将作为 Hadoop 集群中的节点。

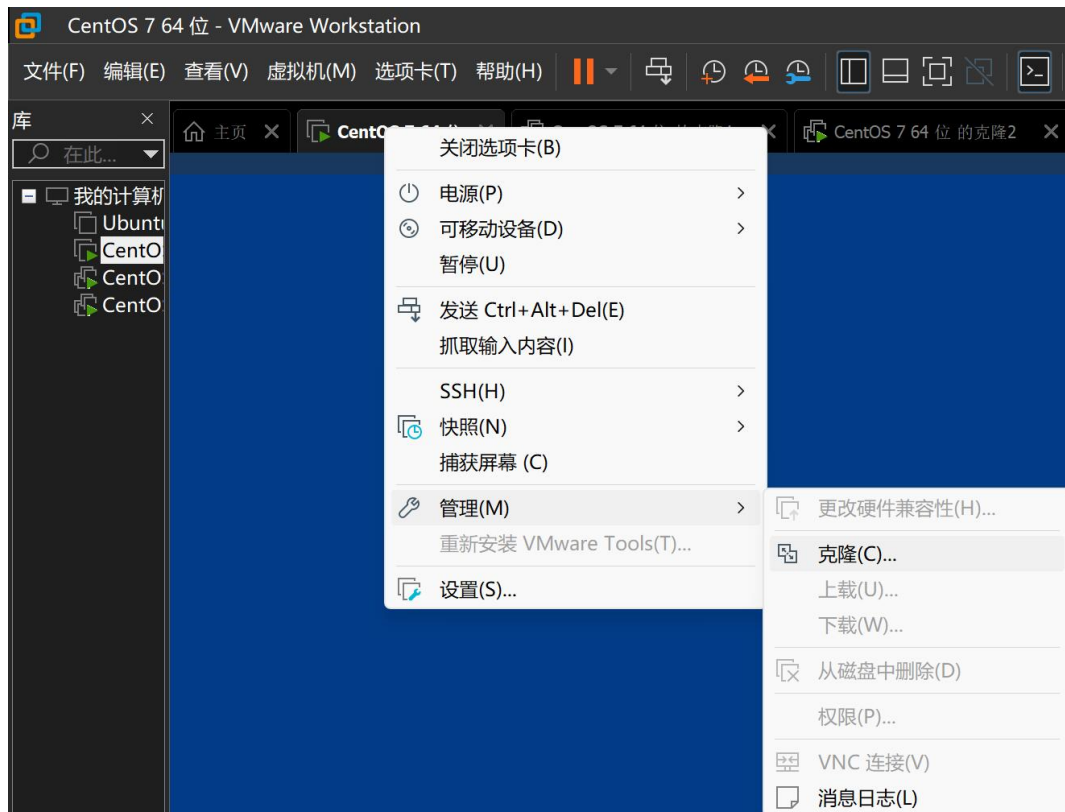


图 1 VMware 克隆虚拟机操作界面

**2. 查看与配置 IP 地址** 进入虚拟机后，使用以下命令查看分配的 IP 地址：

```
1 # 查看网卡信息
2 ifconfig
3
4 # 或者使用 ip 命令
5 ip addr
```

如需设置静态 IP，需修改网卡配置文件（以 `ifcfg-ens33` 为例）：

```
1 TYPE=Ethernet
2 BOOTPROTO=static
3 NAME=ens33
```

```
4 DEVICE=ens33
5 ONBOOT=yes
6
7 IPADDR=192.168.44.139
8 NETMASK=255.255.255.0
9 GATEWAY=192.168.44.2
10 DNS1=223.5.5.5
11 DNS2=8.8.8.8
```

修改完成后，执行以下命令重启网络服务使配置生效：

```
1 systemctl restart network
```

### 3. 设置主机名与 hosts 文件 为保证节点间互通，在每台虚拟机上设置唯一主机名：

```
1 # 在 node1 上执行
2 hostnamectl set-hostname node1
3
4 # 在 node2 上执行
5 hostnamectl set-hostname node2
6
7 # 在 node3 上执行
8 hostnamectl set-hostname node3
```

在所有节点的 /etc/hosts 文件中添加如下映射：

```
1 192.168.44.139 node1
2 192.168.44.140 node2
3 192.168.44.141 node3
```

在 Windows 宿主机的 C:\Windows\System32\drivers\etc\hosts 文件中，也需写入相同映射，确保通过主机名可以直接访问虚拟机。

### 4. 配置 SSH 免密登录 为方便分布式管理，需要配置 SSH 免密。首先在 node1 上生成密钥对：

```
1 ssh-keygen -t rsa
```

将生成的公钥分发至各节点：

```
1 ssh-copy-id node1
2 ssh-copy-id node2
3 ssh-copy-id node3
```

配置完成后，可以直接通过 `ssh node2` 或 `ssh node3` 登录，而无需输入密码。

**5. 验证网络互通** 最后，通过 `ping` 和 `ssh` 验证各节点的连通性：

```
1 # 测试 node1 与 node2 的连通性
2 ping node2
3
4 # 从 node1 免密登录到 node3
5 ssh node3
```

若所有节点均能互通，则虚拟机与网络配置完成，可以进入 Hadoop 集群安装步骤。

## 2.2 Hadoop 集群配置

以下展示对核心配置文件的修改，并解释每个配置项的意义。

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
3 <configuration>
4     <!-- 指定HDFS的NameNode地址 -->
5     <property>
6         <name>fs.defaultFS</name>
7         <value>hdfs://node1:9000</value>
8         <description>NameNode的地址</description>
9     </property>
10
11     <!-- 指定Hadoop临时目录 -->
12     <property>
13         <name>hadoop.tmp.dir</name>
14         <value>/opt/hadoop/tmp</value>
15         <description>Hadoop临时文件存储目录</description>
```

```
16     </property>
17
18     <!-- 缓冲区大小配置 -->
19     <property>
20         <name>io.file.buffer.size</name>
21         <value>131072</value>
22         <description>文件缓冲区大小</description>
23     </property>
24 </configuration>
```

### 代码 1 core-site.xml 核心配置

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
3 <configuration>
4     <!-- 数据副本数量 -->
5     <property>
6         <name>dfs.replication</name>
7         <value>3</value>
8         <description>HDFS数据块副本数量</description>
9     </property>
10
11     <!-- NameNode数据存储路径 -->
12     <property>
13         <name>dfs.namenode.name.dir</name>
14         <value>/opt/hadoop/data/namenode</value>
15         <description>NameNode元数据存储目录</description>
16     </property>
17
18     <!-- DataNode数据存储路径 -->
19     <property>
20         <name>dfs.datanode.data.dir</name>
21         <value>/opt/hadoop/data/datanode</value>
22         <description>DataNode数据存储目录</description>
23     </property>
24
25     <!-- 数据块大小设置 -->
```



```
26     <property>
27         <name>dfs.blocksize</name>
28         <value>134217728</value>
29         <description>HDFS 数据块大小 (128MB)</description>
30     </property>
31 </configuration>
```

### 代码 2 hdfs-site.xml 存储配置

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <configuration>
3     <!-- ResourceManager 地址 -->
4     <property>
5         <name>yarn.resourcemanager.hostname</name>
6         <value>node1</value>
7         <description>ResourceManager 主机名</description>
8     </property>
9
10    <!-- NodeManager 辅助服务 -->
11    <property>
12        <name>yarn.nodemanager.aux-services</name>
13        <value>mapreduce_shuffle</value>
14        <description>NodeManager 辅助服务</description>
15    </property>
16
17    <!-- 内存分配设置 -->
18    <property>
19        <name>yarn.nodemanager.resource.memory-mb</name>
20        <value>2048</value>
21        <description>NodeManager 可用内存</description>
22    </property>
23 </configuration>
```

### 代码 3 yarn-site.xml 资源管理配置

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <configuration>
```

```
3      <!-- 指定MapReduce框架为yarn -->
4      <property>
5          <name>mapreduce.framework.name</name>
6          <value>yarn</value>
7          <description>MapReduce 运行框架</description>
8      </property>
9
10     <!-- JobHistory Server地址 -->
11     <property>
12         <name>mapreduce.jobhistory.address</name>
13         <value>node1:10020</value>
14         <description>历史服务器地址</description>
15     </property>
16
17     <!-- JobHistory Web地址 -->
18     <property>
19         <name>mapreduce.jobhistory.webapp.address</name>
20         <value>node1:19888</value>
21         <description>历史服务器Web界面</description>
22     </property>
23 </configuration>
```

代码 4 mapred-site.xml 作业配置

最后，在 \$HADOOP\_HOME/etc/hadoop/workers 文件中指定工作节点：

```
1 node1
2 node2
3 node3
```

代码 5 workers 配置

## 2.3 HBase 集群配置

在完成 Hadoop 集群部署之后，需要进一步安装并配置 HBase，使其能够依赖 HDFS 进行存储，并借助 ZooKeeper 进行分布式协调。

**1. 解压与环境变量配置** 在所有节点下载并解压 HBase 安装包,然后在 /etc/profile 中配置 HBase 环境变量:

```
1 export HBASE_HOME=/opt/hbase-2.1.0
2 export PATH=$PATH:$HBASE_HOME/bin
```

保存后执行 `source /etc/profile` 使配置立即生效。

**2. 修改 hbase-site.xml** 进入 \$HBASE\_HOME/conf 目录,编辑 hbase-site.xml 文件,主要配置如下:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <configuration>
3   <!-- HBase 在 HDFS 上的存储目录 -->
4   <property>
5     <name>hbase.rootdir</name>
6     <value>hdfs://node1:9000/hbase</value>
7   </property>
8
9   <!-- HBase 以分布式模式运行 -->
10  <property>
11    <name>hbase.cluster.distributed</name>
12    <value>true</value>
13  </property>
14
15  <!-- ZooKeeper 集群地址 -->
16  <property>
17    <name>hbase.zookeeper.quorum</name>
18    <value>node1,node2,node3</value>
19  </property>
20
21  <!-- Write-Ahead Log (WAL) 存储方式 -->
22  <property>
23    <name>hbase.wal.provider</name>
24    <value>filesystem</value>
25  </property>
26
27  <!-- HBase 临时目录 -->
```

```
28     <property>
29         <name>hbase.tmp.dir</name>
30         <value>/opt/hbase/tmp</value>
31     </property>
32 </configuration>
```

代码 6 hbase-site.xml 配置示例

**3. 修改 regionservers 文件** 在 \$HBASE\_HOME/conf/regionservers 文件中，列出所有参与 HBase 的工作节点：

```
1 node1
2 node2
3 node3
```

代码 7 regionservers 配置

**4. hbase-env.sh 配置** 在 hbase-env.sh 文件中，指定 HBase 的 Java 环境：

```
1 export JAVA_HOME=/usr/java/jdk1.8.0_241
```

**5. ZooKeeper 与 HBase 关系说明** HBase 内部依赖 ZooKeeper 进行协调，但若集群已单独部署 ZooKeeper，可以通过上述配置直接指定外部 ZooKeeper 集群地址。否则，HBase 会默认使用自带的 ZooKeeper。

**6. 验证配置** 完成上述配置后，在 node1 上执行：

```
1 start-hbase.sh
```

然后在 node1 上运行：

```
1 hbase shell
```

若能进入 HBase Shell，说明配置成功。之后可以通过 list、create、scan 等命令进一步验证 HBase 的功能。

## 2.4 集群启动与验证

在完成 Hadoop、ZooKeeper 与 HBase 的配置后，需要按顺序启动各个服务，并验证其是否正常运行。

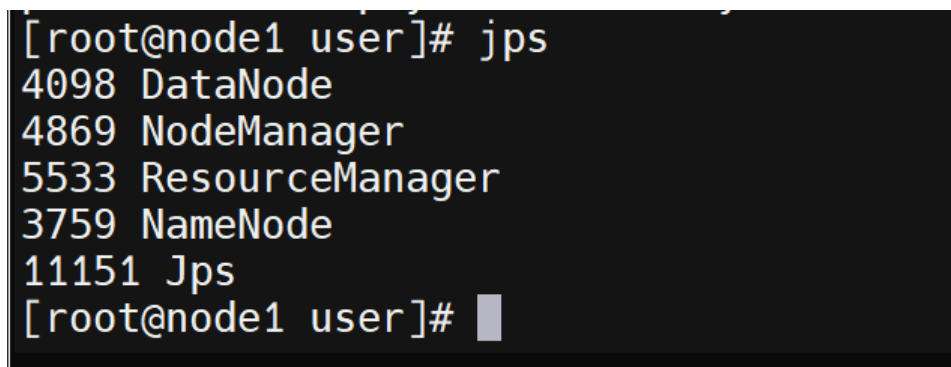
**1. 启动 HDFS 与 YARN** 首先在 node1 节点执行以下命令启动 HDFS 与 YARN 服务：

```
1 # 启动 HDFS
2 start-dfs.sh
3
4 # 启动 YARN
5 start-yarn.sh
```

执行完成后，可通过以下命令检查进程是否正常：

```
1 jps
```

在正常情况下，NameNode、SecondaryNameNode、DataNode、ResourceManager、NodeManager 等进程应全部启动。



```
[root@node1 user]# jps
4098 DataNode
4869 NodeManager
5533 ResourceManager
3759 NameNode
11151 Jps
[root@node1 user]#
```

图 2 node1 节点 jps 进程验证结果（包含 NameNode、ResourceManager 等）

```
[root@node2 user]# jps
4848 Jps
4057 NodeManager
3358 DataNode
3758 SecondaryNameNode
[root@node2 user]#
```

图 3 node2 节点 jps 进程验证结果（包含 SecondaryNameNode、NodeManager 等）

```
[root@node3 user]# jps
12183 Jps
11007 DataNode
11663 NodeManager
[root@node3 user]#
```

图 4 node3 节点 jps 进程验证结果（包含 DataNode、NodeManager 等）

**2. 启动 ZooKeeper 集群** 在 node1、node2、node3 节点分别执行以下命令，启动 ZooKeeper 服务：

```
1 zkServer.sh start
```

使用以下命令查看 ZooKeeper 状态：

```
1 zkServer.sh status
```

其中 node1 显示为 leader，node2 与 node3 显示为 follower 即为正常。

**3. 启动 HBase** 在 node1 上执行以下命令启动 HBase 服务：

```
1 start-hbase.sh
```

随后可通过以下命令验证进程：

```
1 jps
```

若看到 HMaster 与 HRegionServer 等进程，说明 HBase 启动成功。

```
[root@node1 user]# jps
39713 Jps
4098 DataNode
4869 NodeManager
27895 HRegionServer
26088 QuorumPeerMain
27690 HMaster
5533 ResourceManager
3759 NameNode
[root@node1 user]#
```

图 5 node1 节点 jps 进程验证结果（包含 HMaster、HRegionServer 等）

```
[root@node2 user]# jps
6757 Jps
4057 NodeManager
3358 DataNode
3758 SecondaryNameNode
5743 QuorumPeerMain
5935 HRegionServer
[root@node2 user]#
```

图 6 node2 节点 jps 进程验证结果（包含 SecondaryNameNode、HRegionServer 等）

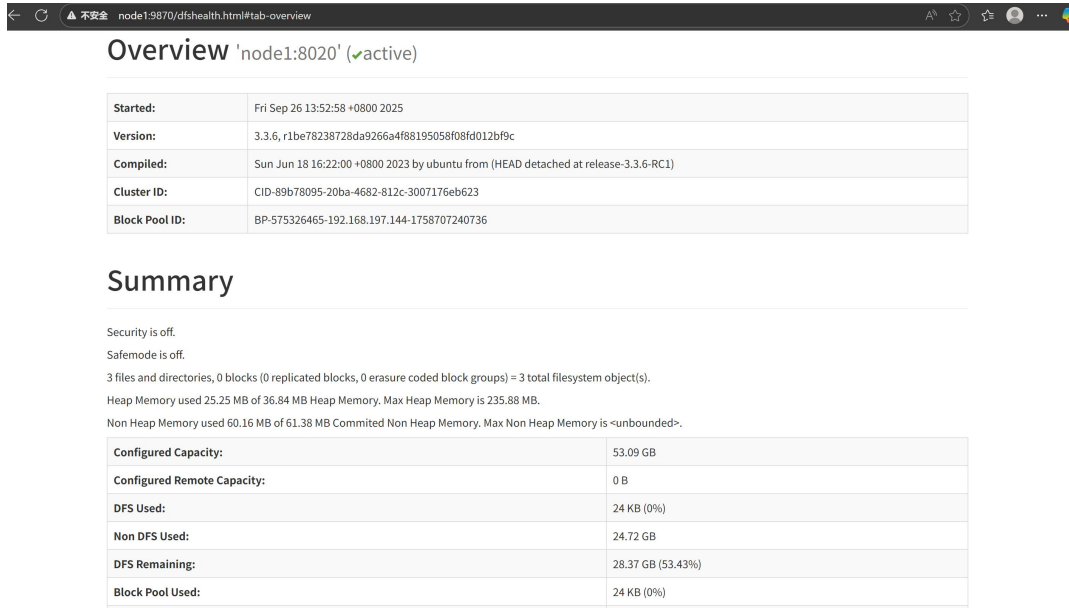
```
[root@node3 user]# jps
13187 HRegionServer
13019 QuorumPeerMain
13965 Jps
11007 DataNode
11663 NodeManager
[root@node3 user]#
```

图 7 node3 节点 jps 进程验证结果（包含 HRegionServer 等）

**4. Web UI 验证** 各组件启动成功后，可以通过 Web 界面进行验证：

- HDFS NameNode 管理界面: <http://node1:9870>
- YARN ResourceManager 界面: <http://node1:8088>
- HBase Master 界面: <http://node1:16010>

通过浏览器访问上述地址, 可以查看各个服务的运行状态及任务执行情况。



**Overview 'node1:8020' (✓active)**

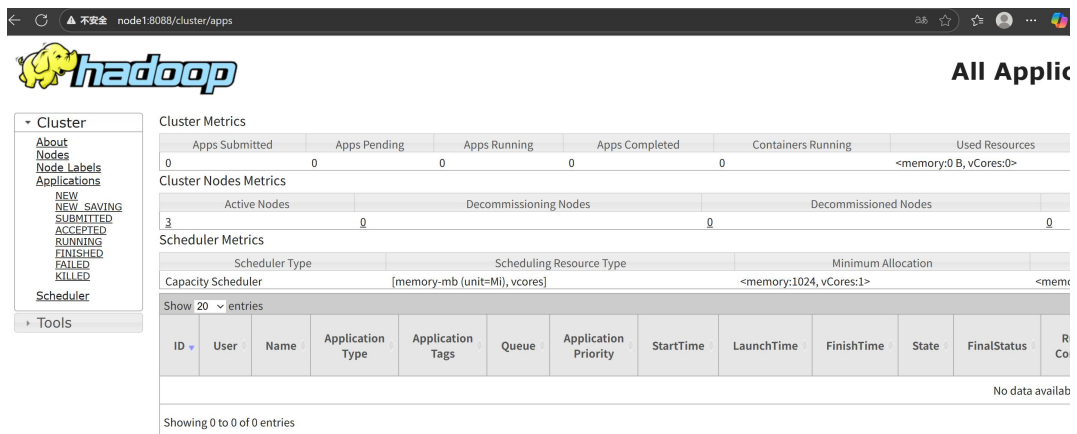
Started:	Fri Sep 26 13:52:58 +0800 2025
Version:	3.3.6, r1be78238728da9266a4f88195058f08fd012bf9c
Compiled:	Sun Jun 18 16:22:00 +0800 2023 by ubuntu from (HEAD detached at release-3.3.6-RC1)
Cluster ID:	CID-89b78095-20ba-4682-812c-3007176eb623
Block Pool ID:	BP-575326465-192.168.197.144-1758707240736

**Summary**

Security is off.  
Safemode is off.  
3 files and directories, 0 blocks (0 replicated blocks, 0 erasure coded block groups) = 3 total filesystem object(s).  
Heap Memory used 25.25 MB of 36.84 MB Heap Memory. Max Heap Memory is 235.88 MB.  
Non Heap Memory used 60.16 MB of 61.38 MB Committed Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	53.09 GB
Configured Remote Capacity:	0 B
DFS Used:	24 KB (0%)
Non DFS Used:	24.72 GB
DFS Remaining:	28.37 GB (53.43%)
Block Pool Used:	24 KB (0%)

图 8 HDFS NameNode Web 界面 (<http://node1:9870>)



**Cluster Metrics**

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Used Resources
0	0	0	0	0	<memory:0 B, vCores:0>

**Cluster Nodes Metrics**

Active Nodes	Decommissioning Nodes	Decommissioned Nodes
3	0	0

**Scheduler Metrics**

Scheduler Type	Scheduling Resource Type	Minimum Allocation
Capacity Scheduler	[memory-mb (unit=M), vcores]	<memory:1024, vCores:1>

Show 20 entries

ID	User	Name	Application Type	Application Tags	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus	Resource
No data available												

Showing 0 to 0 of 0 entries

图 9 YARN ResourceManager Web 界面 (<http://node1:8088>)



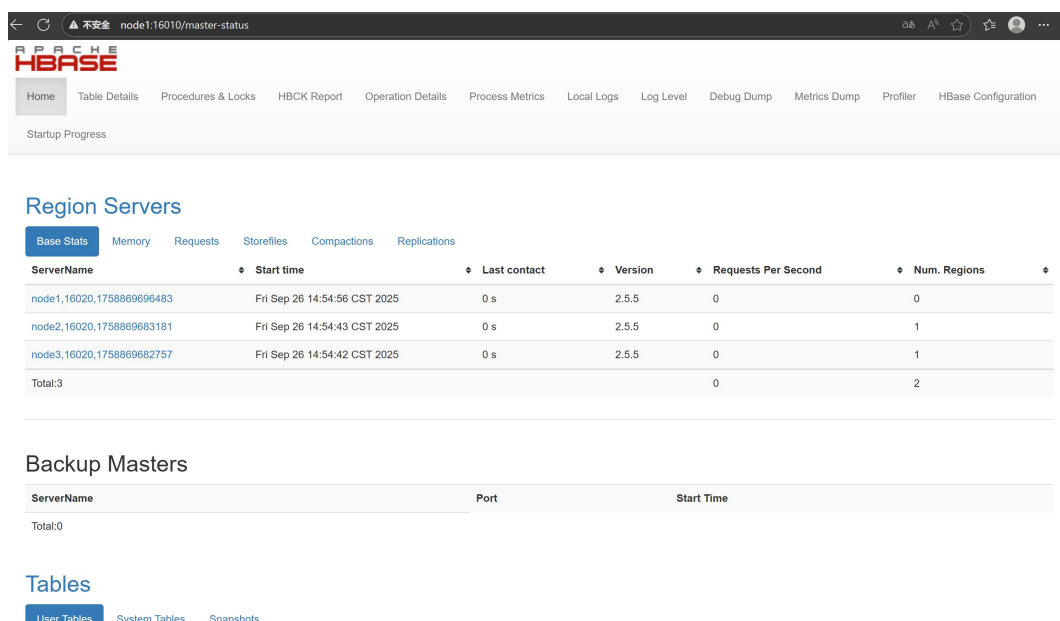


图 10 HBase Master Web 界面 (http://node1:16010)

5. HBase Shell 验证 最后进入 HBase Shell，执行以下命令验证功能：

```
1 hbase shell
2
3 # 在 HBase Shell 中输入
4 list
```

若能够正确输出当前已有的表，说明 HBase 与 Hadoop 集群已成功运行。

### 3. 实验过程与实现

#### 3.1 数据准备

本次实验所使用的原始数据集为 `sentences.txt`，大小约为 1.43GB，数据量较大。如果直接将该文件作为 MapReduce 的输入，单节点读取和处理的效率较低。因此，在实验开始前，我们首先对原始文件进行分割处理，使其能够更好地并行分布到各个节点进行计算。

为了实现数据分割，我们编写了一个 Python 脚本，按照每 10000 行划分为一个子文件。其核心代码如下所示：

```
1 def split_file(input_filename, lines_per_file=10000):
2     # 打开输入文件
3     with open(input_filename, 'r', encoding='utf-8') as input_file:
4         file_count = 1      # 输出文件计数器
5         current_output = None
6
7         for line_number, line in enumerate(input_file, 1):
8             # 每当开始新文件时, 关闭旧文件并创建新文件
9             if line_number % lines_per_file == 1:
10                 if current_output:
11                     current_output.close()
12                     output_filename = f"split_{file_count}.txt"
13                     current_output = open(output_filename, 'w',
14                                           encoding='utf-8')
15                     file_count += 1
16                 # 将当前行写入输出文件
17                 current_output.write(line)
18
19             if current_output:
20                 current_output.close()
21
22 # 使用函数对 sentences.txt 进行分割
23 split_file('./sentences.txt')
```

### 代码8 文件分割脚本

运行该脚本后, 原始文件被拆分为多个大小约为 1.4MB 的子文件, 如图 11 所示。每个子文件包含约 10000 行数据, 最终共生成 940 个分片文件, 分别命名为 split\_1.txt 至 split\_940.txt。

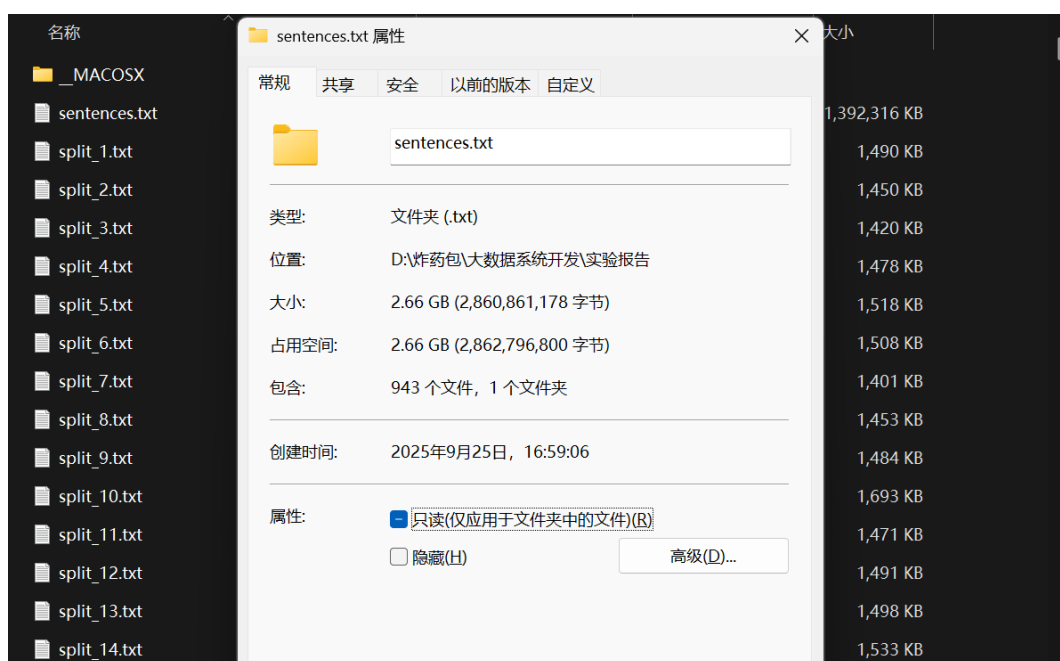


图 11 数据集拆分结果

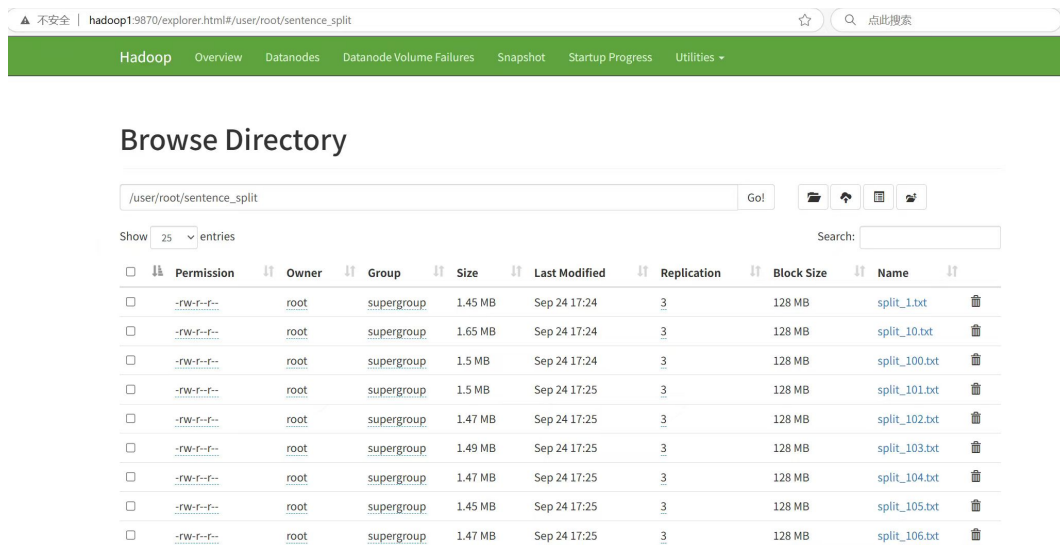
接下来，需要将这些拆分后的文件上传到 HDFS，以便后续的 MapReduce 程序进行处理。首先，在 HDFS 中创建一个用于存放实验输入数据的目录：

```
1 hdfs dfs -mkdir -p /user/input
```

然后，将拆分好的文件批量上传至该目录：

```
1 hdfs dfs -put split_*.txt /user/input
```

至此，实验所需的数据已经准备完毕，HDFS 中的目录结构如图 12 所示。



Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	root	supergroup	1.45 MB	Sep 24 17:24	3	128 MB	split_1.txt
-rw-r--r--	root	supergroup	1.65 MB	Sep 24 17:24	3	128 MB	split_10.txt
-rw-r--r--	root	supergroup	1.5 MB	Sep 24 17:24	3	128 MB	split_100.txt
-rw-r--r--	root	supergroup	1.5 MB	Sep 24 17:25	3	128 MB	split_101.txt
-rw-r--r--	root	supergroup	1.47 MB	Sep 24 17:25	3	128 MB	split_102.txt
-rw-r--r--	root	supergroup	1.49 MB	Sep 24 17:25	3	128 MB	split_103.txt
-rw-r--r--	root	supergroup	1.47 MB	Sep 24 17:25	3	128 MB	split_104.txt
-rw-r--r--	root	supergroup	1.45 MB	Sep 24 17:25	3	128 MB	split_105.txt
-rw-r--r--	root	supergroup	1.47 MB	Sep 24 17:25	3	128 MB	split_106.txt

图 12 上传至 HDFS 的数据目录

## 3.2 核心算法与代码实现

### 3.2.1 Mapper 实现

Mapper 的核心任务是对输入的文本数据进行分词，并记录单词出现的位置信息。在本实验中，位置信息不仅包括单词本身，还包含该单词所在的文件名，从而为后续构建倒排索引做好准备。

```
1 public static class Map extends Mapper<Object, Text, Text, Text> {
2     private Text keyInfo = new Text();
3     private Text valueInfo = new Text();
4     private FileSplit split;
5
6     public void map(Object key, Text value, Context context)
7         throws IOException, InterruptedException {
8         split = (FileSplit) context.getInputSplit();
9         StringTokenizer itr = new StringTokenizer(value.toString());
10        while (itr.hasMoreTokens()) {
11            // 获取当前处理的文件名
12            String fileName = split.getPath().getName();
13            // 将单词与文件名拼接作为key
14            keyInfo.set(itr.nextToken() + ":" + fileName);
15            // 值固定为1，表示出现一次
```

```
16         valueInfo.set("1");
17         // 输出<单词:文件名, 1>
18         context.write(keyInfo, valueInfo);
19     }
20 }
21 }
```

### 代码 9 Mapper 核心代码

在上述实现中，map() 函数的处理逻辑如下：

- 通过 FileSplit 获取当前正在处理的文件名，用于标识单词的来源文件。
- 使用 StringTokenizer 对输入行进行分词，逐个提取单词。
- 将单词与文件名拼接形成新的键 (word:filename)，对应的值统一设置为 1，表示该单词在该文件中出现了一次。
- 通过 context.write() 输出键值对，交由框架传递给后续的 Combine 或 Reduce 阶段。

这样处理的好处在于：在 Map 阶段就已经将单词与所在文件建立了联系，为后续统计和去重操作奠定了基础。最终目标是能够在 HBase 中形成倒排索引表，使得任意一个单词可以快速定位到其所在的文件集合。

### 3.2.2 Reducer 实现

Reducer 的主要任务是将 Map 阶段输出的 < 单词，文件编号 > 进行汇总，最终生成倒排索引，并写入到 HBase 表中。不同于传统的 MapReduce 将结果输出到 HDFS，本实验通过 TableReducer 类直接将结果存储到 HBase 的指定列族和列中。

```
1 public static class InvertedIndexReducer
2     extends TableReducer<Text, Text, ImmutableBytesWritable> {
3
4     @Override
5     protected void reduce(Text key, Iterable<Text> values, Context
        context)
```

```
6         throws IOException, InterruptedException {
7         // 用于存储所有文件编号
8         List<String> fileList = new ArrayList<>();
9
10        for (Text v : values) {
11            String fileNum = v.toString();
12            // 避免重复, 将相同文件编号只保存一次
13            if (!fileList.contains(fileNum)) {
14                fileList.add(fileNum);
15            }
16        }
17
18        // 将结果拼接成以逗号分隔的字符串
19        StringBuilder sb = new StringBuilder();
20        for (String f : fileList) {
21            sb.append(f).append(",");
22        }
23        // 去掉最后一个多余的逗号
24        String result = sb.substring(0, sb.length() - 1);
25
26        // 构建Put对象: RowKey 为单词
27        Put put = new Put(Bytes.toBytes(key.toString()));
28        // 写入列族col_family, 列名info
29        put.addColumn(Bytes.toBytes("col_family"),
30                      Bytes.toBytes("info"),
31                      Bytes.toBytes(result));
32
33        // 输出到HBase
34        context.write(null, put);
35    }
36 }
```

### 代码 10 Reducer 核心代码

在上述实现中，reduce() 函数的核心逻辑如下：

- 接收来自 Map 阶段的所有值（文件编号），这些值表示某个单词出现过的文件。

- 使用 `List<String>` 对文件编号进行去重，避免重复计入同一文件。
- 将所有文件编号拼接成一个以逗号分隔的字符串，例如 `'1,3,5'`，表示该单词出现在 1、3、5 号文件中。
- 构造 HBase 的 `Put` 对象，以单词作为行键（`RowKey`），在列族 `col_family` 下的列 `info` 中写入拼接后的文件编号列表。
- 通过 `context.write()` 方法将结果直接写入 HBase 表，实现单词到文件编号集合的倒排索引存储。

与传统 MapReduce 输出到 HDFS 不同，本实验使用 `TableReducer` 将结果直接与 HBase 集成。这不仅简化了数据落地的流程，还为后续的数据查询和分析提供了高效的支持：只需在 HBase 中查询某个单词对应的行键，即可快速获得该单词出现过的所有文件编号。

### 3.2.3 Driver 实现

Driver 部分是整个 MapReduce 程序的入口，用于配置作业运行环境、指定 Mapper 与 Reducer 类、设置输入输出路径等。本实验的 Driver 还负责调用 HBase 提供的工具类，使 MapReduce 的结果直接写入到 HBase 表中。

```
1 public static void main(String[] args)
2     throws IOException, ClassNotFoundException, InterruptedException {
3
4     // 1. 创建 Hadoop 与 HBase 的配置对象
5     Configuration conf = HBaseConfiguration.create();
6
7     // 2. 解析命令行参数，获取输入路径
8     String[] otherArgs = new GenericOptionsParser(conf, args)
9         .getRemainingArgs();
10
11     // 3. 创建 Job 实例
12     Job job = Job.getInstance(conf, "InvertedIndex");
13     job.setJarByClass(InvertedIndex.class);
14 }
```

```
15 // 4. 设置 Mapper 与 Reducer
16 job.setMapperClass(MyMapper.class);
17 job.setReducerClass(MyReducer.class);
18
19 job.setMapOutputKeyClass(Text.class);
20 job.setMapOutputValueClass(Text.class);
21
22 // 5. 将输出结果写入到 HBase 表 test_table
23 TableMapReduceUtil.initTableReducerJob(
24     "test_table",          // HBase 表名
25     MyReducer.class,      // Reducer 类
26     job
27 );
28
29 // 6. 指定输入路径 (HDFS 上的数据文件)
30 FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
31
32 // 7. 提交任务并等待执行完成
33 System.exit(job.waitForCompletion(true) ? 0 : 1);
34 }
```

### 代码 11 Driver 核心代码

在上述代码中，Driver 的关键逻辑如下：

1. 使用 `HBaseConfiguration.create()` 生成配置对象，从而支持与 HBase 的集成。
2. 通过 `GenericOptionsParser` 解析命令行参数，获取 MapReduce 输入数据所在的 HDFS 路径。
3. 设置作业的核心类 `InvertedIndex`，并指定对应的 Mapper 与 Reducer 类。
4. 调用 `TableMapReduceUtil.initTableReducerJob()`，将 Reducer 的输出直接写入到 HBase 表 `test_table`，而不是传统的 HDFS 文件。
5. 通过 `FileInputFormat.addInputPath()` 指定输入路径，从而让 Hadoop 在 HDFS 中读取拆分后的语料文件。



6. 最后调用 `job.waitForCompletion(true)` 提交作业并阻塞等待执行结果。

通过这种方式，本实验实现了从 HDFS 输入 → MapReduce 计算 → HBase 存储的完整大数据处理流程。相比于将结果写回 HDFS 文件，再导入数据库的方式，这种集成方案大大简化了处理链路，体现了 Hadoop 与 HBase 的紧密结合。

### 3.3 程序打包与执行

在完成核心代码编写后，需要将项目打包为 JAR 文件，以便提交给 Hadoop 集群执行。本实验采用 **Maven** 进行项目管理与打包，执行以下命令即可生成可运行的 JAR 包：

```
1 mvn clean package -DskipTests
```

上述命令会在 `target/` 目录下生成 `InvertedIndex-1.0-SNAPSHOT.jar` 文件，该文件即为我们提交到集群的作业包。

随后，使用 Hadoop 命令行提交 MapReduce 作业，指定输入目录和输出表：

```
1 hadoop jar target/InvertedIndex-1.0-SNAPSHOT.jar \
2     InvertedIndex \
3     /user/input \
4     /user/output
```

其中：

- `InvertedIndex` 为主类名。
- `/user/input` 为 HDFS 中存放的分割后语料目录。
- `/user/output` 为 MapReduce 的临时输出路径（即使最终结果写入 HBase，也需要设置）。

运行过程中，Hadoop 会将作业分发到各个节点并输出执行日志。若作业成功完成，可以在 HBase 中查询到倒排索引的存储结果。

## 4. 实验结果与分析

### 4.1 运行过程监控

```

2025-09-27 23:22:37,566 INFO mapreduce.Job: The url to track the job: http://hadoop1:8088/proxy/application_1758981452611_0002/
2025-09-27 23:22:37,567 INFO mapreduce.Job: Running job: job_1758981452611_0002
2025-09-27 23:23:08,768 INFO mapreduce.Job: Job job_1758981452611_0002 running in uber mode : false
2025-09-27 23:23:08,825 INFO mapreduce.Job: map 0% reduce 0%
2025-09-27 23:24:13,500 INFO mapreduce.Job: map 1% reduce 0%
2025-09-27 23:24:18,629 INFO mapreduce.Job: map 13% reduce 0%
2025-09-27 23:24:23,881 INFO mapreduce.Job: map 18% reduce 0%
2025-09-27 23:24:33,426 INFO mapreduce.Job: map 20% reduce 0%
2025-09-27 23:24:41,939 INFO mapreduce.Job: map 23% reduce 0%
2025-09-27 23:24:45,498 INFO mapreduce.Job: map 24% reduce 0%
2025-09-27 23:24:47,827 INFO mapreduce.Job: map 25% reduce 0%
2025-09-27 23:24:51,230 INFO mapreduce.Job: map 27% reduce 0%
2025-09-27 23:24:52,263 INFO mapreduce.Job: map 31% reduce 0%
2025-09-27 23:24:56,690 INFO mapreduce.Job: map 31% reduce 8%
2025-09-27 23:24:57,746 INFO mapreduce.Job: map 34% reduce 8%
2025-09-27 23:24:58,796 INFO mapreduce.Job: map 35% reduce 8%
2025-09-27 23:24:59,854 INFO mapreduce.Job: map 40% reduce 8%
2025-09-27 23:25:01,996 INFO mapreduce.Job: map 40% reduce 12%
2025-09-27 23:25:04,327 INFO mapreduce.Job: map 46% reduce 12%
2025-09-27 23:25:05,387 INFO mapreduce.Job: map 51% reduce 12%
2025-09-27 23:25:06,431 INFO mapreduce.Job: map 53% reduce 12%
2025-09-27 23:25:07,467 INFO mapreduce.Job: map 58% reduce 12%
2025-09-27 23:25:08,493 INFO mapreduce.Job: map 60% reduce 18%
2025-09-27 23:25:14,588 INFO mapreduce.Job: map 60% reduce 20%
2025-09-27 23:25:15,613 INFO mapreduce.Job: map 65% reduce 20%
2025-09-27 23:25:20,706 INFO mapreduce.Job: map 65% reduce 22%
2025-09-27 23:25:26,813 INFO mapreduce.Job: map 70% reduce 22%
2025-09-27 23:25:27,825 INFO mapreduce.Job: map 71% reduce 22%
2025-09-27 23:25:29,858 INFO mapreduce.Job: map 72% reduce 22%
2025-09-27 23:25:30,875 INFO mapreduce.Job: map 76% reduce 22%
2025-09-27 23:25:31,890 INFO mapreduce.Job: map 78% reduce 22%
2025-09-27 23:25:32,903 INFO mapreduce.Job: map 78% reduce 23%
2025-09-27 23:25:34,956 INFO mapreduce.Job: map 79% reduce 23%
2025-09-27 23:25:37,023 INFO mapreduce.Job: map 85% reduce 23%
2025-09-27 23:25:38,036 INFO mapreduce.Job: map 91% reduce 23%
2025-09-27 23:25:40,071 INFO mapreduce.Job: map 94% reduce 23%
2025-09-27 23:25:41,094 INFO mapreduce.Job: map 97% reduce 23%
2025-09-27 23:25:42,130 INFO mapreduce.Job: map 100% reduce 23%
2025-09-27 23:25:45,211 INFO mapreduce.Job: map 100% reduce 67%
2025-09-27 23:25:56,469 INFO mapreduce.Job: map 100% reduce 70%
2025-09-27 23:26:02,650 INFO mapreduce.Job: map 100% reduce 91%
2025-09-27 23:26:04,686 INFO mapreduce.Job: map 100% reduce 100%
2025-09-27 23:26:08,854 INFO mapreduce.Job: Job job_1758981452611_0002 completed successfully
2025-09-27 23:26:10,648 INFO mapreduce.Job: Counters: 55
File System Counters

```

图 13 运行进度显示

可以看到，MapReduce 作业已经成功完成，所有的 map 任务和 reduce 任务均已成功执行。

### 4.2 结果验证

```

1 hbase shell
2 create 'InvertedIndexTable', 'fileInfo'
3 list

```

```
hbase(main):001:0> list
TABLE
InvertedIndexTable
1 row(s)
Took 1.2054 seconds
=> ["InvertedIndexTable"]
```

图 14 HBase Shell 中 list 命令查询表结果

可以看到，表 InvertedIndexTable 已经成功创建。

```
scan 'InvertedIndexTable', {STARTROW => 'good', LIMIT => 10}
```

```
hbase(main):005:0> scan 'InvertedIndexTable', {STARTROW => 'good', LIMIT => 10}
ROW COLUMN+CELL
good COLUMN=timestamp=1758986760377, value=split_10.txt:291;split_16.txt:341;split_3.txt:321;split_1.txt:395;split_19.txt:275;split_17.txt:301;split_11.txt:393;split_15.txt:353;split_20.txt:273;split_7.txt:381;split_18.txt:316;split_14.txt:354;split_12.txt:359;split_13.txt:310;split_6.txt:334;split_4.txt:321;split_8.txt:347;split_5.txt:345;split_9.txt:358;split_2.txt:338;
goodale COLUMN=timestamp=1758986760377, value=split_18.txt:1;
goodbody COLUMN=timestamp=1758986760377, value=split_17.txt:1;
goodbye COLUMN=timestamp=1758986760377, value=split_17.txt:1;split_2.txt:2;split_3.txt:2;split_10.txt:1;split_1.txt:1;split_19.txt:1;split_14.txt:5;split_20.txt:1;split_15.txt:2;split_7.txt:6;split_9.txt:8;split_6.txt:1;split_12.txt:1;split_13.txt:2;split_5.txt:6;split_4.txt:2;split_8.txt:3;
goodell COLUMN=timestamp=1758986760377, value=split_4.txt:1;
gooderness COLUMN=timestamp=1758986760377, value=split_1.txt:1;
goodesy COLUMN=timestamp=1758986760377, value=split_15.txt:1;
goodte COLUMN=timestamp=1758986760377, value=split_9.txt:3;
goodies COLUMN=timestamp=1758986760377, value=split_17.txt:1;split_18.txt:1;split_16.txt:2;split_15.txt:1;split_6.txt:1;
goodts COLUMN=timestamp=1758986760377, value=split_9.txt:1;split_12.txt:1;
10 row(s)
Took 0.6632 seconds
```

图 15 HBase Shell 中 scan 命令查询以'good'开头的单词结果

可以看到，以单词'good'开头的单词及其所在的文件编号已经成功存入 HBase 表中。

```
1 get 'InvertedIndexTable', 'hello'
2 get 'InvertedIndexTable', 'world'
```

```
hbase(main):002:0> get 'InvertedIndexTable', 'hello'
COLUMN CELL
fileInfo:fileList timestamp=1758986760523, value=split_1.txt:2;split_20.txt:6;split_5.txt:1;split_7.txt:1;split_13.txt:7;split_19.txt:22;split_14.txt:13;split_18.txt:6;split_2.txt:1;split_12.txt:11;split_9.txt:3;split_17.txt:9;split_10.txt:8;split_4.txt:4;split_8.txt:9;split_3.txt:3;split_16.txt:21;split_15.txt:17;split_11.txt:1;split_6.txt:4;
1 row(s)
Took 0.4092 seconds
hbase(main):003:0> get 'InvertedIndexTable', 'world'
COLUMN CELL
fileInfo:fileList timestamp=1758986763132, value=split_8.txt:210;split_20.txt:176;split_10.txt:239;split_4.txt:202;split_9.txt:229;split_15.txt:181;split_3.txt:197;split_14.txt:182;split_11.txt:209;split_7.txt:192;split_1.txt:195;split_2.txt:145;split_19.txt:191;split_18.txt:236;split_6.txt:216;split_17.txt:191;split_5.txt:171;split_13.txt:184;split_12.txt:190;split_16.txt:167;
1 row(s)
Took 0.0143 seconds
```

图 16 HBase Shell 中 get 命令查询结果

可以看到，单词'hello'和单词'world'均成功存入 HBase 表中，并且其所在的文件编号也正确无误。

### 4.3 性能分析

本次实现在伪分布式和完全分布式两种模式下均进行了测试。我们搭建的伪分布式环境为一台机器完成所有任务，而完全分布式环境共有三台机器，一台机器作为 namenode，另外两台机器作为 datanode。

该 mapreduce 任务分别在两种环境下运行，记录任务的执行时间。伪分布式模式下，任务执行时间为 143min，而在完全分布式模式下，任务执行时间只需 54min。

在小数据集下，伪分布式模式与完全分布式模式的差异并不明显，但随着数据量的增大，完全分布式模式的高效率，高性能优势将会愈发明显，从而体现出分布式计算的强大能力。

由此看出，分布式计算能够有效提升大数据处理的效率，充分发挥多节点协同工作的优势。

## 5. 遇到的问题及解决方案

在运行 mapreduce 作业时，reduce 任务因执行超时，被应用程序管理器终止，进而导致任务失败。

```
AttemptID:attempt_1758803251058_0001_r_000000_1 task timeout set: 600s, taskTimedOut: true;
task stuck timeout set: 600s, taskStuck: false
```

查看日志发现暂停来自主机层面的资源瓶颈。磁盘分区使用率达到了 89 %，导致系统写入临时文件失败、应用程序因缺乏磁盘空间无法正常运行。

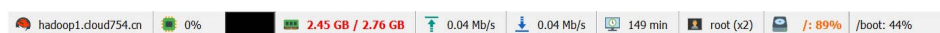


图 17 磁盘使用率过高

查阅资料后，发现主节点任务多，可以通过增大硬盘容量来解决该问题。命令行操作如下：

```
1 sudo fdisk -l # 查看当前磁盘分区
2 sudo fdisk /dev/sdb # 选择需要扩展的磁盘
3 partprobe /dev/sda # 让内核重新读取分区表
4 pvcreate /dev/sda3 # 创建物理卷
5 vgextend centos /dev/sda3 # 扩展卷组
6 lvextend /dev/centos/root /dev/sda3 # 扩展逻辑卷
7 resize2fs /dev/centos/root # 扩展文件系统
8 df -h # 查看扩展结果
```

最后得到：/dev/mapper/centos-root 31G 15G 17G 46% 可见磁盘扩展成功，问题得到解决。

## 6. 总结与心得

### 6.1 实验总结

本次实验是一次较为完整的大数据系统开发实践，从分布式环境搭建到应用程序实现再到结果验证，涵盖了大数据处理的主要环节。通过这次实验，我们不仅验证了 Hadoop 生态系统的可行性和强大功能，还进一步加深了对分布式计算与存储模式的理解。

在环境搭建方面，我们首先利用 VMware 平台克隆虚拟机，建立了一个三节点的分布式实验环境。在每个节点上，我们手动配置了静态 IP 地址、主机名和 `/etc/hosts` 文件映射，并设置 SSH 免密登录，确保节点之间能够顺畅通信。这些操作虽然繁琐，但却让我们对分布式系统的运行依赖有了更加直观的体会：一个小小的配置错误，例如 IP 地址冲突或主机名映射遗漏，都可能导致整个集群无法启动。这一过程强化了我们在工程实践中“细节决定成败”的认识。

在框架部署方面，我们成功安装并配置了 Hadoop、ZooKeeper 和 HBase。通过修改 `core-site.xml`、`hdfs-site.xml`、`yarn-site.xml` 等核心配置文件，我们明确了 HDFS 的存储目录、副本数量以及 YARN 的资源调度参数。随后配置的 ZooKeeper 集群为分布式协调提供了可靠支撑，而 HBase 的安装与配置使得我们能够基于 HDFS 存储实现高效的 NoSQL 查询。经过一系列调试与测试，最终集群能够稳定运行，并支持大数据存储与计算任务。

在应用实现方面，我们基于 MapReduce 编程模型完成了倒排索引的构建。通过对原始数据集进行分片处理，使任务能够被合理地分配到不同节点执行。在 Mapper 阶段，我们完成了文本分词与键值对的生成；在 Reducer 阶段，则对分词结果进行聚合统计，并最终写入 HBase 数据库中。实验过程中，我们通过 HDFS Web UI、YARN 任务管理界面以及 HBase Shell 进行了结果验证，确认了倒排索引的正确性与完整性。

在实验结果分析中，我们还对比了伪分布式与完全分布式环境下任务执行时间的差异。结果表明，随着节点数量的增加，MapReduce 任务的执行效率显著提升，充分体现了分布式架构在可扩展性与性能提升方面的优势。这一结果不仅符合我们对分布式系统的理论认知，也进一步证明了 Hadoop 生态在大规模数据处理中具有重要应用

价值。

总的来说，本次实验实现了从系统搭建到应用开发的全流程验证，最终顺利完成了倒排索引的构建任务。更为重要的是，实验让我们真正理解了大数据系统各个组件之间的协作关系，以及分布式计算环境下进行应用开发的挑战与价值，为后续进一步研究与实践打下了坚实基础。

## 6.2 心得体会

通过本次实践，我们在分布式计算的理解、系统调试的能力以及团队协作的意识上都有了明显的提升。整个实验的过程不仅仅是简单地完成一系列任务，更是一种综合能力的训练。

在技术层面上，我们对分布式计算原理的理解得到了加深。实验中小组成员亲身经历了从单机环境到多节点集群的转变，体会到 MapReduce 编程模型“分而治之”的思想。Mapper 负责对大规模数据进行分割处理，Reducer 负责结果的聚合与统计，这种任务拆解方式显著提升了计算效率。通过运行实际任务，我们感受到分布式系统的核心价值：并行性、扩展性与容错性。在真实环境下，即使某个节点出现问题，集群依旧能够通过副本机制保证数据的可靠性，这使我们对分布式系统的健壮性有了更深刻的体会。

在工程实践层面上，实验让我们认识到系统配置和调试的重要性。起初我们遇到多种问题，例如节点间 SSH 无法免密登录、ZooKeeper 配置不一致导致服务无法启动、HBase 无法正常连接 HDFS 等。这些问题迫使我们仔细检查每一个配置文件，并通过日志分析、逐步排错的方式找到并解决问题。这一过程虽然耗时，但极大提升了我们独立解决复杂问题的能力，也让我们学会了如何在工程实践中保持耐心与细心。

在团队协作层面上，本次实验同样带来了深刻的收获。由于实验涉及的环节较多，从虚拟机环境的搭建、集群的配置，到 MapReduce 程序的实现与调试，每个阶段都需要小组成员之间的紧密配合。我们通过合理分工、及时沟通与相互检查，保证了实验能够顺利推进。这让我们意识到，分布式系统的学习与开发不仅仅是技术问题，更是协作与组织问题。良好的沟通与分工机制，往往能在关键时刻提高效率，避免重复劳动和错误。

综合来看，本次实验不仅帮助我们掌握了 Hadoop、ZooKeeper 和 HBase 的基本使用方法，还加深了对大数据系统整体架构的理解。在未来的学习与科研中，这些宝贵的经验将为我们应对更加复杂的工程实践提供坚实的支持。同时，实验中形成的系统化思维方式和团队合作意识，也将成为我们小组在后续研究和开发工作中的重要财富。