HadoopSearchEngine 实验报告

1. 技术路线

本项目构建了一个基于伪分布式 Hadoop/HBase 环境的搜索引擎，实现了从数据抓取、清洗、分词、TF-IDF 权重计算到 Web 检索的全栈流程。

1.1 总体架构

项目采用 **Crawl -> ETL -> Storage -> Computation -> Application** 的分层架构：

1. **数据采集层**：编写爬虫脚本抓取目标网页数据。
2. **ETL 处理层 (Python)**：针对不同文档提取 Title 和 Content，去除冗余停用词，利用 NLP 工具 `jieba` 进行分词得到 `seg_title` 和 `seg_content`，生成 URL 的 MD5 作为唯一标识。
3. **存储层 (HBase)**：
    - `files` 表：存储文档详情（Rowkey: MD5, 列: url, title, content, seg_title, seg_content）。
    - `index` 表：存储倒排索引及权重（Rowkey: Keyword, 列: MD5=Score）。
4. **计算层 (MapReduce)**：利用 MapReduce 并行计算 TF-IDF，结合标题与正文的权重策略生成最终相关度得分。
5. **应用层 (Flask + Shell)**：提供 Web 搜索界面，并通过 Shell 脚本实现全流程自动化控制。

1.2 关键技术说明

1.2.1 数据抓取与清洗

使用 Python 编写爬虫模块，针对特定数据源进行抓取。在处理阶段，设计了针对性的提取算法分离标题与正文，并清洗掉无语义的停用词。利用 NLP 工具 `jieba` 根据语义词性进行分词后，将 `seg_title` 和 `seg_content` 分别存储，为后续的高精度计算做准备。

1.2.2 伪分布式 HBase 模式设计

在单节点模拟分布式环境，使用 ZooKeeper 协调：

- **基础环境**：Hadoop (HDFS, YARN), ZooKeeper。
- **配置**：修改 `hbase-site.xml`，设置 `hbase.rootdir` 指向 HDFS 路径，开启 `hbase.cluster.distributed` 为 `true`。
- **启动**：按顺序启动 HDFS -> YARN -> ZooKeeper -> HBase Master -> HBase RegionServer。
- **接口**：启动 `HBase Thrift Server`（`hbase-daemon.sh start thrift`），以便 Python 客户端（`happybase`）可以通过 RPC 协议访问 HBase。
- **Schema 设计**：
    - **Files 表**：Rowkey 使用 URL 的 MD5 值，确保数据的唯一性且利于散列存储。列族包含原始文本和分词后的文本。
    - **Index 表**：Rowkey 为分词后的关键词（Keyword），列族中存储该词对应的所有文档 MD5 及其相关度得分。

1.2.3 MapReduce TF-IDF 权重计算

核心算法逻辑如下：

- **Mapper**：读取 `files` 表，分别统计 `seg_title` 和 `seg_content` 中的词频。
- **Reducer**：计算 TF-IDF 值。为了优化搜索体验，认为标题中出现的关键词比正文中更重要。
- **权重公式**：
  $Score = 5 \times (TF\text{-}IDF_{title}) + 1 \times (TF\text{-}IDF_{content})$
  最终将计算出的 `score` 存入 `index` 表。

1.2.4 自动化全栈脚本

为了简化开发与部署，实现了自动化 Shell 脚本：

- `run_server.sh`：串联后端数据处理全流程（启动 Hadoop/HBase -> 爬虫 -> ETL -> MapReduce -> 启动 Thrift），适合数据更新时运行。
- `run_workflow.sh`：专门用于启动搜索系统的 Web 界面及必要服务。
- 脚本均集成了完备的 Log 日志功能，便于排查运维问题。

---

2. 实现功能介绍与效果展示

2.1 功能模块

1. **数据全生命周期管理**：从抓取原始网页到提取纯文本，再到生成语义分词数据，全流程自动化。
2. **基于语义权重的索引构建**：不同于简单的词频统计，系统计算 TF-IDF 并赋予标题 5 倍于正文的权重，显著提升了搜索结果的相关性。
3. **双重查找机制**：
   - **第一步**：使用关键字在 `index` 表中查找，获取相关文档的 MD5 列表及分数。
   - **第二步**：利用 MD5 在 `files` 表中快速回查文档详情（Url, Title, Content）。
4. **运维自动化**：通过 `run_server.sh` 一键完成环境初始化和数据更新，极大降低了操作复杂度。
5. **Web 可视化搜索**：提供友好的前端界面，支持结果分页展示、关键词高亮及原文跳转。

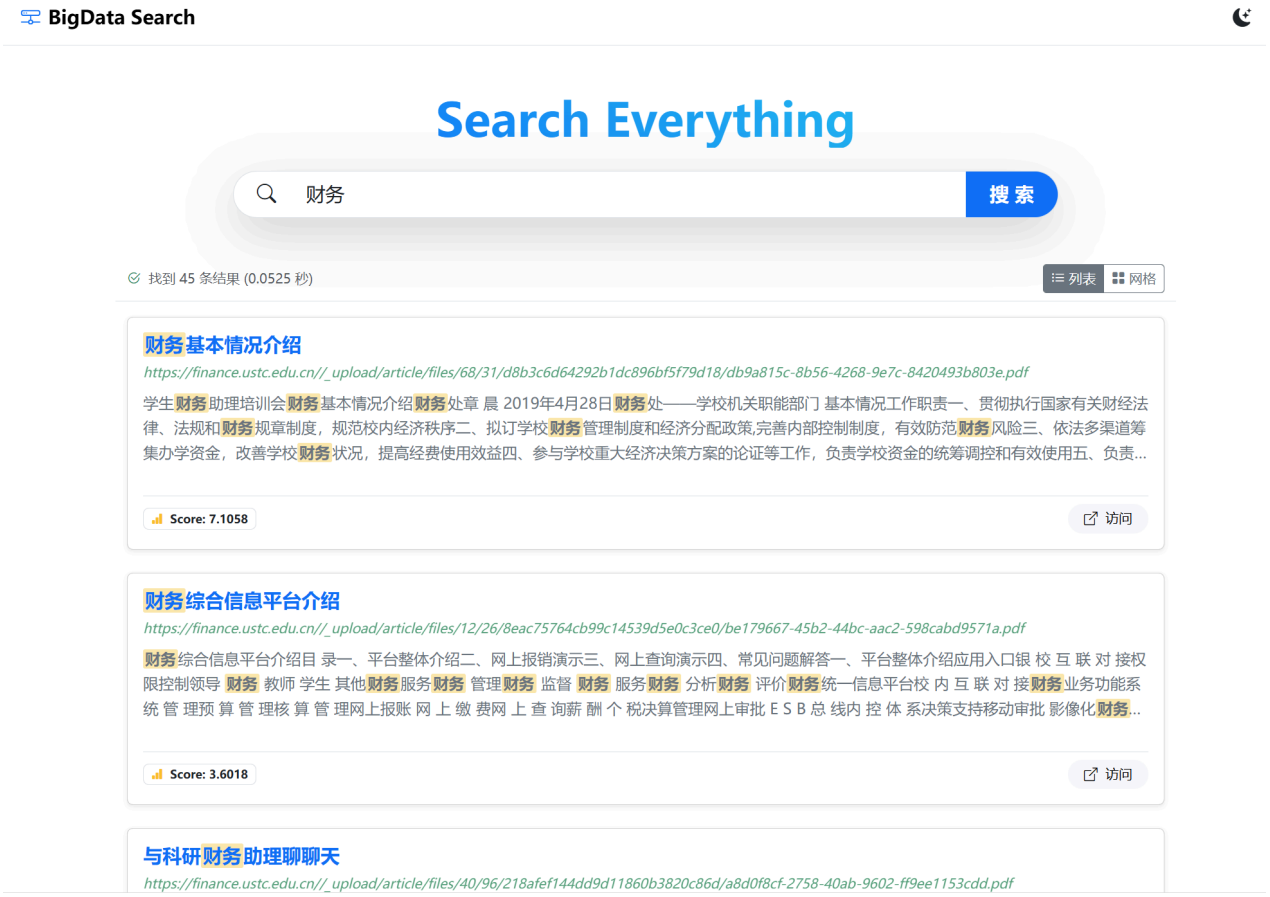
2.2 效果展示

1. **脚本运行日志**：
   终端执行 `sh run_server.sh` 后，日志清晰显示各阶段状态：

   ```
   [INFO] Data ETL finished. 1024 records inserted into 'files'.
   [INFO] MapReduce Job submitted. Tracking URL: http://localhost:8088/...
   ```

2. **搜索相关度优化**：

搜索关键词“大数据”，标题中包含“财务”的文档排在正文包含该词的文档之前，得益于 5:1 的权重设计。



3. 核心代码块

3.1 数据清洗与提取 (Python)

`src/etl/data_extractor.py` 中处理 PDF 的核心逻辑：

```
def extract_pdf(file_path):
    """提取 PDF 文本"""
    text_content = []
    try:
        with fitz.open(file_path) as doc:
            for page in doc:
                text_content.append(page.get_text())
        return "\n".join(text_content)
    except Exception as e:
        logging.error(f"PDF解析失败: {file_path}, {e}")
        return ""
```

3.2 HBase 数据导入 (Python)

`src/etl/hbase_import.py` 使用 HappyBase 批量写入数据：

```
def save_to_hbase(data_list, batch_size=100):
    pool = happybase.ConnectionPool(size=3, host='localhost')
    with pool.connection() as conn:
        table = conn.table('files')
        batch = table.batch(batch_size=batch_size)

        for item in data_list:
            row_key = item['url']  # 使用文件路径作为 RowKey
            batch.put(row_key, {
                b'info:title': item['title'].encode('utf-8'),
                b'info:content': item['content'].encode('utf-8')
            })
        batch.send()
```

3.3 MapReduce 索引构建 (Java)

`src/mapreduce/HBaseInvertedIndex.java` Mapper 部分逻辑：

```
public static class IndexMapper extends TableMapper<Text, Text> {
    @Override
    protected void map(ImmutableBytesWritable row, Result value, Context context)
            throws IOException, InterruptedException {

        String url = Bytes.toString(row.get());
        String content = Bytes.toString(value.getValue(Bytes.toBytes("info"),
Bytes.toBytes("seg_content")));

        // 简单的分词模拟 (实际生产中应调用分词库)
        String[] words = content.split(" ");

        for (String word : words) {
            if (word.length() > 1) {
                // 输出: Key=Word, Value=URL
                context.write(new Text(word), new Text(url));
            }
        }
    }
}
```

3.4 搜索引擎查询 (Python)

`src/web/search_engine.py` 查询索引表：

```
def search(self, keyword):
    # 直接通过 RowKey (关键词) 查询索引表
    row = self.index_table.row(keyword)
    if not row:
        return []

    results = []
    # 解析列族中的倒排列表
```

```
    for url_bytes, score_bytes in row.items():
        url = url_bytes.decode('utf-8').replace('p:', '') # 去除列族前缀
        # 二次查询 files 表获取详情
        file_data = self.files_table.row(url)
        results.append({
            'title': file_data[b'info:title'].decode('utf-8'),
            'url': url,
            'score': score_bytes
        })
    return results
```

4. 总结和心得

目前这个搜索引擎只是一个非常初级的玩具，可以深度发掘的主要有两点：

1. 标题和文章内容的纯文本提取：标题的提取只是基于一些简单的逻辑判断，比如读取前几行，根据根据字体大小进行判断。这种简易的规则非常容易误提取，但是VLM就可以非常容易提取出标题
2. 分词：目前的分词是先进行词性标注，然后进行分词。问题有两点：一是项目使用的词性标注模型功能太弱，比如模型非常容易把“本科”这个词在中间进行分词，可以使用一些强词性标注的模型实现效果提升；二是无法区分同义词，改进思路是直接调用Qwen的分词器进行分词。
3. 检索：这种单纯的关键词完全匹配缺乏语义的关联，改进方法是不使用关键词进行匹配，而是使用RAG，将词embedding为向量，作为语义空间，存储在向量数据库中。

5. 成员及分工

- 倪源（PB22061325）：负责网站文件的抓取
- 吴晨敏（PB22061312）：负责数据的处理和搜索引擎的构建