热词统计与分析系统设计文档

课程名称：	数据结构与算法	任课教师：	张子臻

年级:	2024级	专业(方向):	计算机科学与技术(人工智能与大数据)
学号:	24325158	姓名:	梁宇聪

一、系统概述

- **项目背景**：针对互联网海量文本流，设计一个能够实时监控、统计指定时间窗口内高频词汇（Top-K）的系统。
- **核心目标**：实现基于时间的滑动窗口算法，支持 O(1) 级别的实时更新，并提供交互式的可视化查询界面。
- **技术栈**：Python 3.11, jieba (分词), streamlit + altair (可视化), pytest (测试)。

二、背景假设与外部依赖

(一) 输入数据假设

1. **Auto模式**：输入数据按照 `[H:MM:SS] <文本数据>` 的格式提供，每行一条文本数据。在需要查询的位置插入一行 `[ACTION] QUERY K=<整数>` 。如：

```
[0:04:27] 你们三兄弟长的一点都不一样
[ACTION] QUERY K=6
[0:04:27] 诸葛瑾确实是神级管家！其他能力就一般了
```

2. **Interactive模式**：输入的文本数据同样按照 `[H:MM:SS] <文本数据>` 的格式提供，每行一条文本数据。但不再需要在文件中提供 `[ACTION] QUERY K=<整数>` 指令。用户可以在终端交互窗口处手动输入时间点和窗口大小进行查询

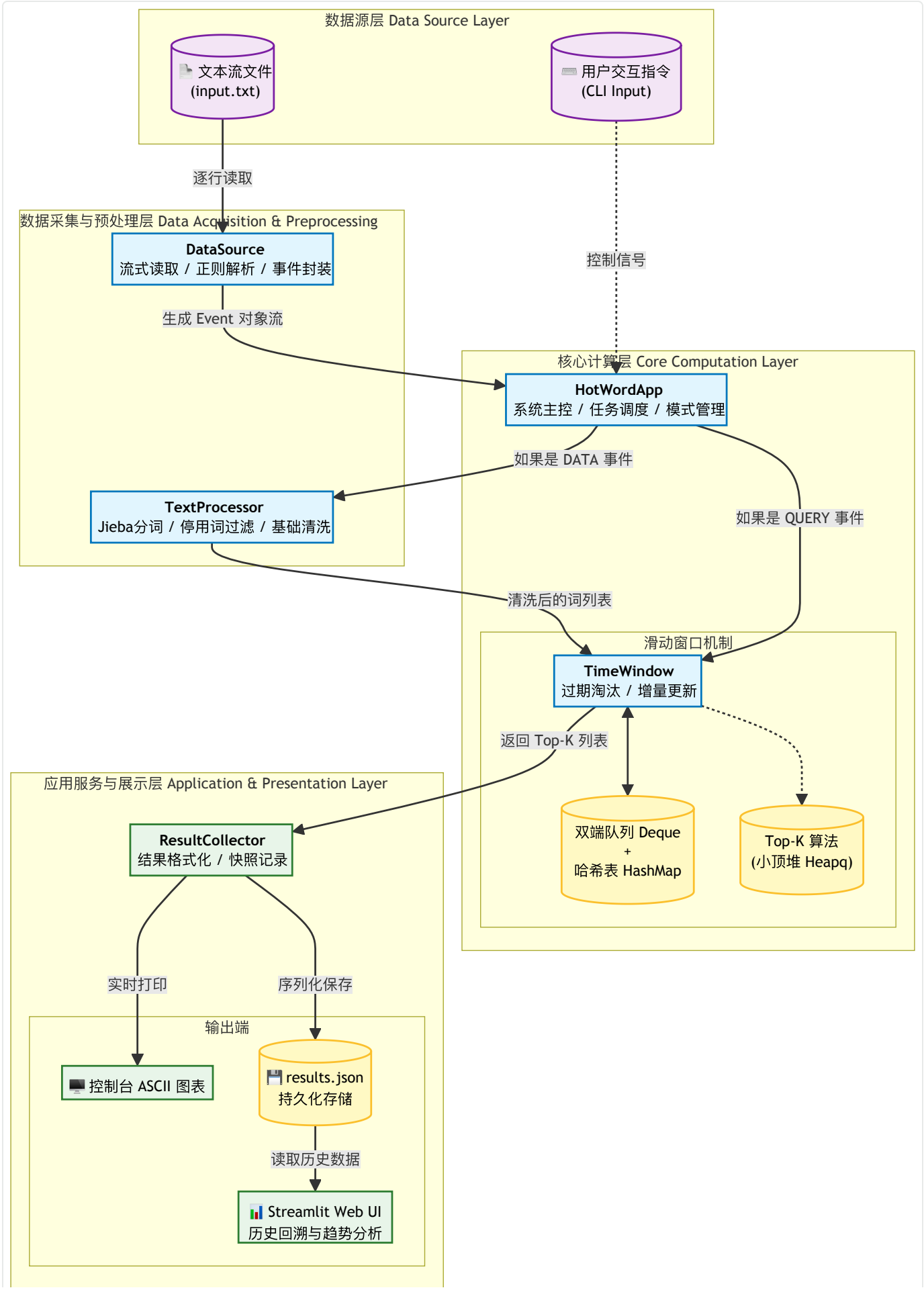
注：输入文件均使用 UTF-8 编码

(二) 外部依赖

- **Jieba**: 用于中文分词。
  - 版本：0.42.1
  - 许可证：MIT License

- **Streamlit**：用于构建 Web 界面。
  - 版本：1.52.2
  - 许可证：Apache 2.0
- **altair**：用于实现多彩柱状图
  - 版本：6.0.0
  - 许可证：BSD 3-Clause
- **pytest**：用于进行单元测试
  - 版本：9.0.2
  - 许可证：MIT

三、系统架构设计


数据源层 Data Source Layer
文本流文件
(input.txt)
用户交互指令
(CLI Input)
逐行读取
控制信号
数据采集与预处理层 Data Acquisition & Preprocessing
DataSource
流式读取 / 正则解析 / 事件封装
生成 Event 对象流
核心计算层 Core Computation Layer
HotWordApp
系统主控 / 任务调度 / 模式管理
如果是 DATA 事件
TextProcessor
Jieba分词 / 停用词过滤 / 基础清洗
如果是 QUERY 事件
清洗后的词列表
滑动窗口机制
TimeWindow
过期淘汰 / 增量更新
返回 Top-K 列表
双端队列 Deque
+
哈希表 HashMap
Top-K 算法
(小顶堆 Heapq)
应用服务与展示层 Application & Presentation Layer
ResultCollector
结果格式化 / 快照记录
实时打印
序列化保存
输出端
控制台 ASCII 图表
results.json
持久化存储
读取历史数据
Streamlit Web UI
历史回溯与趋势分析

本系统采用**分层架构**设计模式，将数据采集、预处理、核心计算与应用服务进行解耦，以满足流式数据处理的高内聚、低耦合要求。系统自下而上主要分为四个逻辑层次：

(一) 数据源层 (Data Source Layer)

- **功能描述**：负责对接外部输入数据流，屏蔽底层文件读取细节。
- **设计实现**：采用生成器（Generator）模式逐行读取模拟数据源（文本文件），有效降低了大规模数据处理时的内存占用。该层通过正则表达式解析原始文本，提取时间戳、文本内容或控制指令（如QUERY），将其封装为统一的事件对象（Event）向上层传递。

(二) 数据预处理层 (Data Preprocessing Layer)

- **功能描述**：负责对原始文本进行清洗和标准化，为统计核心提供高质量的输入。
- **设计实现**：集成了中文分词工具（Jieba），实现了停用词过滤（Stopwords Filtering）和非法字符清洗。该层确保只有有意义的、长度合规的词汇进入下游统计模块。

(三) 核心计算层 (Core Computation Layer)

- **功能描述**：这是系统的核心层，实现了基于时间的滑动窗口算法和 Top-K 统计。
- **设计实现**：
  - **窗口管理**：采用 **双端队列（Deque）** 结合 **哈希表（Hash Map）** 的双重结构。队列用于维护词汇的时间顺序，支持过期数据的 $O(1)$ 移除；哈希表用于维护词频，支持 $O(1)$ 的增量更新。
  - **Top-K 查询**：利用 **小顶堆（Min-Heap）** 算法（通过 Python 的 `heapq` 实现）在查询时快速提取频率最高的 K 个词汇，保证了查询效率。

(四) 应用控制与交互层 (Application Control & Presentation Layer)

- **功能描述**：负责系统的整体调度、用户交互及结果的可视化。
- **设计实现**：
  - **控制器**：`HotWordApp` 作为主控类，协调数据流转，支持“自动批处理”和“交互式查询”两种运行模式。
  - **展示与持久化**：支持控制台 ASCII 柱状图的实时绘制，并将查询快照序列化为 JSON 文件。
  - **可视化前端**：独立的 Streamlit 模块读取持久化数据，提供基于 Web 的历史回溯与趋势分析界面。

四、核心数据结构与算法

为了满足题目对**时间复杂度**和**内存占用**的严格约束，本系统在 Python 标准库的基础上进行了以下选型：

(一) 滑动窗口管理

- **设计选择**：采用 **双端队列** 结合 **哈希表** 。
- **Python 实现**： `collections.deque` + `collections.defaultdict` 。
- **设计理由**：
  - 流式数据按时间顺序到达，过期数据需要从窗口中删除，符合队列“先进先出”的特性。
  - `deque` 在两端的 `append` 和 `popleft` 操作均为 $O(1)$ 复杂度，远优于普通列表的 $O(N)$ 。
- **过期策略**：
  - 每当新数据到达时，检查队头元素的时间戳。若 `当前时间 - 队头时间 > 窗口长度` ，则循环执行 `popleft` 并更新哈希表，直至队头合法。
- **代码片段**：

```python
def del_expired_words(self, timestamp):
        """将过期词汇弹出窗口并在哈希表中删除"""
        while self.window:
            head_time, head_word = self.window[0]

            # 如果词语超出了窗口大小，将其在弹出队列并在哈希表中删除
            if timestamp - head_time > self.window_size:
                self.window.popleft()
                self.hash[head_word] -= 1

                # 如果词频已经减为0，将其从哈希表中删除
                if self.hash[head_word] == 0:
                    del self.hash[head_word]

            else:
                break

    def add_word(self, timestamp, word):
        """添加词汇"""
        # 每次添加新词前先将超出窗口时间的词语删除
        self.del_expired_words(timestamp)

        # 保证时间戳合法，合法指的是当前词汇进入窗口的时间不得早于上一行词汇进入窗口的时间，否则认为是错误数据
        if self.is_valid_time(timestamp):
            self.window.append((timestamp, word))
            self.hash[word] += 1
```

(二) Top-K 查询

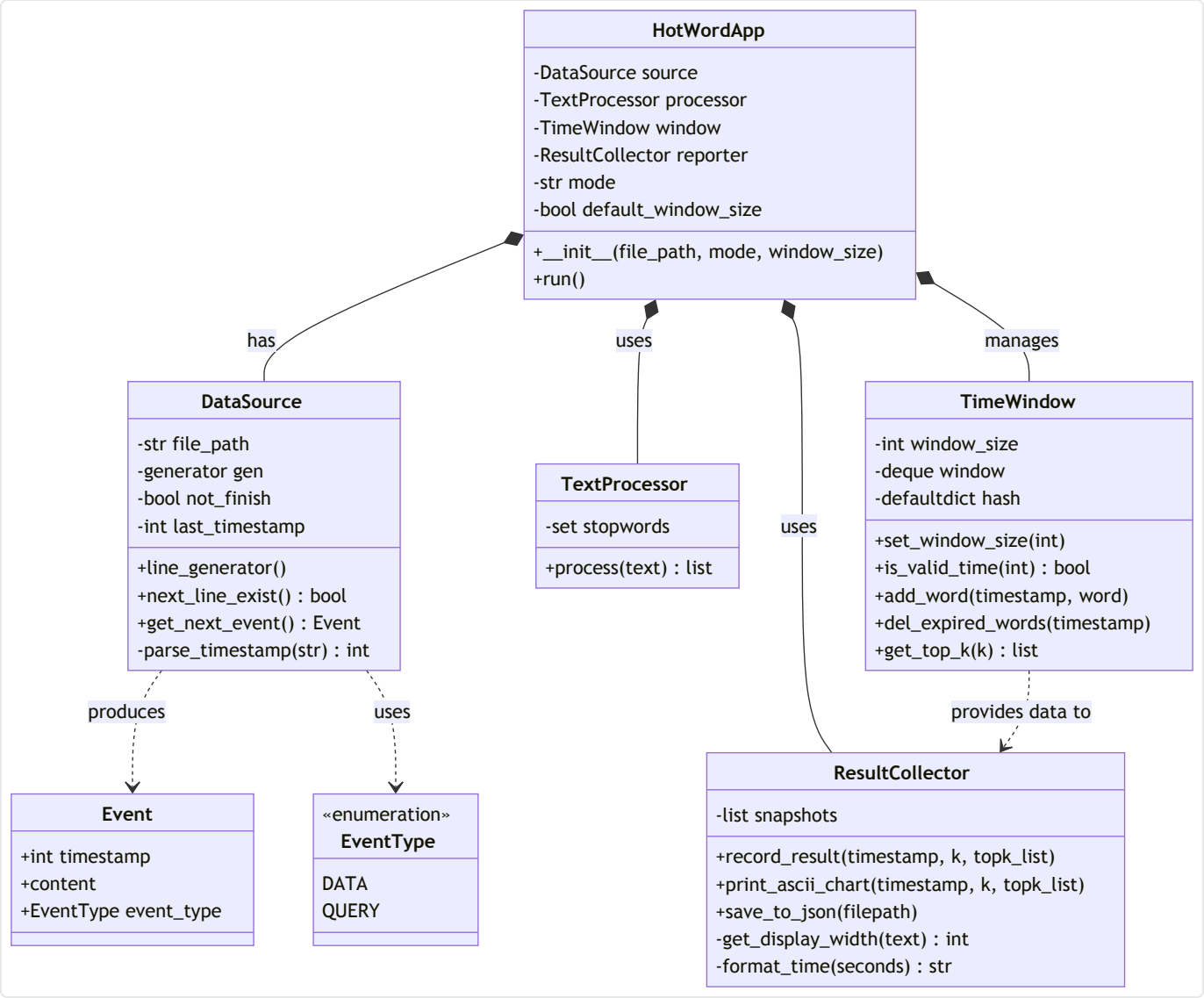
- **设计选择**：**哈希表+ 最小堆选择**。
- **Python 实现**：`heapq.nsmallest`。
- **算法逻辑**：
  - 日常数据流入仅更新哈希表，复杂度 $O(1)$ 。
  - 仅在触发 `QUERY` 指令时，对哈希表进行 Top-K 提取。
- **复杂度分析**：设哈希表中不同词汇数量为 M ，提取 Top-K 的复杂度为 $O(M \log K)$ 。
- **代码片段**：

```
def get_top_k(self, k):
        """求出当前的topK热词"""
        # 出现次数前添加负号使得出现次数和字典序都能按升序排但是结果正确
        return heapq.nsmallest(k, self.hash.items(), key=lambda x: (-x[1], x[0]))
```

(三) 复杂度汇总表

操作	对应算法/结构	时间复杂度	备注
新增词汇	Hash Map Update + Deque Append	$O(1)$	均摊复杂度
过期淘汰	Deque Popleft	$O(1)$	均摊，每个元素仅进出一次
Top-K 查询	Heap Select (nlargest)	$O(M \log K)$	M 为当前窗口内去重词数

五、模块详细设计



本节从面向对象设计的角度，详细说明系统中关键类的职责、数据结构选择及核心算法。

(一) 主控模块（`HotWordApp`）

- **职责**：系统的入口与协调者。
- **关键逻辑**：
  - 维护全剧的时间状态。
  - 根据 `EventType`（DATA 或 QUERY）分发任务：数据事件调用 `process` 和 `add_word`；查询事件调用 `get_top_k` 和 `reporter`。
  - 在交互模式下，负责处理用户输入的时间跳转逻辑，确保时间单调递增。

(二) 数据接入模块（`DataSource`）

- **类名**：`DataSource`

- **职责**：将非结构化的文本流转换为结构化的 `Event` 对象流。
- **关键技术**：
  - 使用 `yield` 关键字实现惰性加载，避免一次性加载大文件导致内存溢出。
  - 正则表达式 `re` 精确匹配 `[HH:MM:SS]` 时间戳格式和 `[ACTION]` 指令，具备一定的容错性。

(三) 核心统计模块 (`TimeWindow`)

- **类名**： `TimeWindow`
- **职责**：实现滑动窗口逻辑，维护当前窗口内的词频状态。
- **核心数据结构**：
  - `self.window (deque)`: 存储 `(timestamp, word)` 元组。选择双端队列是因为滑动窗口不仅需要尾部追加（新词进入），还需要头部弹出（过期词移除），Deque 在两端操作的时间复杂度均为 $O(1)$ 。
  - `self.hash (defaultdict)`: 存储 `{word: count}` 。提供 $O(1)$ 的词频查询与更新。
- **关键算法**：
  - **惰性淘汰策略**：在每次 `add_word` 时，检查队首元素的时间戳。如果 `current_time - head_time > window_size` ，则循环弹出队首并递减哈希表计数。若计数归零，则从哈希表中彻底删除键值，防止内存泄漏。
  - **Top-K 排序**： `get_top_k` 方法使用 `heapq.nsmallest` ，配合 Lambda 表达式 `key=lambda x: (-x[1], x[0])` ，实现了按词频降序、同词频按字典序升序的复合排序要求。

(四) 文本处理模块 (`TextProcessor`)

- **类名**： `TextProcessor`
- **职责**：封装 Jieba 分词库。
- **初始化**：在 `__init__` 中加载 `data/stopwords.txt` 到 `set` 集合中（查找复杂度 $O(1)$ ），用于后续的高效过滤。

(五) 结果报告模块 (`ResultCollector`)

- **类名**： `ResultCollector`
- **职责**：处理输出格式与数据持久化。
- **特色设计**：
  - `get_display_width` ：利用 `unicodedata` 判断字符的全角/半角属性，解决了控制台

输出中文时对齐错乱的问题，实现了美观的 ASCII 柱状图。

- `save_to_json`：将内存中的快照列表序列化存储，实现了计算与展示（Streamlit）的解耦。

六、性能优化与资源评估方法

(一) 资源占用评估方法

为了确保系统在资源受限的环境下稳定运行，采用了以下模型估算内存占用：

$$Memory \approx S_{deque} + S_{dict} + S_{overhead}$$

- **队列占用（S_{deque}）**
  窗口内存储的是 `(timestamp, word)` 元组。Python 中一个 `tuple` 约 48 字节，加上字符串引用的指针开销。
  若窗口内有 10^5 个词，每个词平均长度 4 字符（约 50–60 bytes 对象开销），队列大约占用：$10^5 \times (48 + 8 + 60)\ \text{bytes} \approx 11\ \text{MB}$。
- **哈希表占用（S_{dict}）**
  存储唯一词与频次。Python 的 `dict` 也是稀疏存储，会有一定的内存扩容冗余。若唯一词为 2×10^4 个，占用约 $2 - 5\ \text{MB}$。
- **结论**
  在常规 10 分钟窗口（假设 10w 数据量）下，核心数据结构内存占用控制在 **50MB** 以内，完全符合一般服务器或个人 PC 的内存限制。

(二) 性能优化策略

在本项目中，针对 Python 语言特性及题目要求，实施了以下优化：

1. **生成器模式读取**
   在 `DataSource` 中使用 `yield` 逐行读取文件，而非一次性 `readlines()` 加载内存。这使得系统能够处理 GB 级别的日志文件而不发生内存溢出 (OOM)，实现了真正的**流式处理**。
2. **O(1) 滑动窗口淘汰**
   利用 `collections.deque` 的 FIFO 特性。每次新数据进入时，检查队头数据的时间戳。
   - `popleft()` 操作复杂度为 $O(1)$。
   - 配合哈希表 `hash[word] -= 1`，避免了遍历整个窗口来删除过期数据的开销。
3. **惰性计算与即时更新结合**:
   - 词频更新是**即时**的（写入时更新），保证数据实时性。

- Top-K 排序是**惰性**的（查询时计算），在写多读少的数据流场景下（例如每秒写入1000次，每分钟查询1次），避免了每次写入都维护有序结构的巨大开销。

4. **字符串处理**:
   在 `ResultCollector` 的可视化输出中，预先计算了中文字符的 `east_asian_width`，避免了打印时不对齐的问题，虽然增加了微小的计算量，但提升了用户体验。在正式性能测试中关闭此功能以换取最大吞吐。

七、可视化与交互

(一) 热词统计显示

- 利用 Streamlit 构建 Web 界面。
- 横轴表示热词，纵轴表示频率。
- 引入时间滑块，允许用户按时间轴回溯查看不同时刻的 Top-K 变化趋势。

> 效果如下：

热词统计与分析系统 - 查询历史回溯

通过滑动下方滑块，查看不同查询时刻的 Top-K 热词分布。

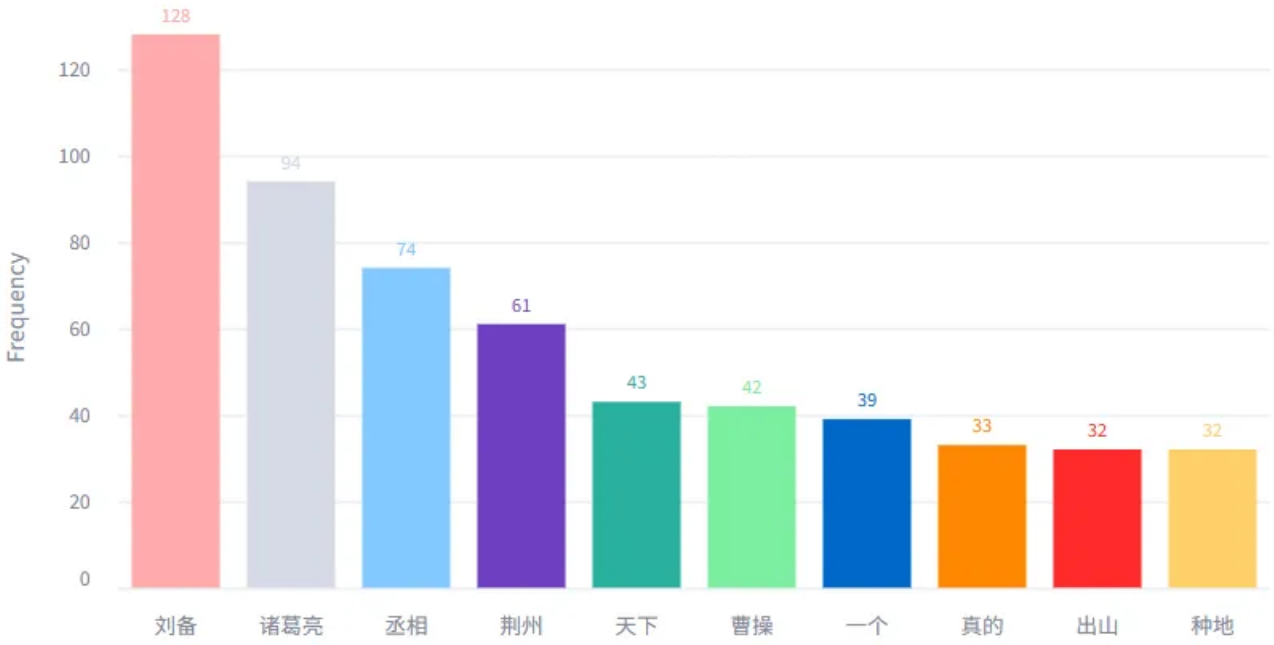
选择第几个查询时刻 (Query Index)

9

1 16

⏱️ 查询时间: 00:41:49

当前查询参数: Top-10



查看详细数据表

	Word	Frequency
0	刘备	128
1	诸葛亮	94
2	丞相	74
3	荆州	61
4	天下	43
5	曹操	42
6	一个	39
7	真的	33
8	出山	32
9	种地	32

(二) 交互模式下终端显示

- 每次查询将结果在终端打印柱形图显示，便于用户实时看到结果
- 交互模式下的历次查询在统计结束后也会在web界面显示

> 效果如下：