中山大学

计算机学院

数据结构与算法大作业

热词统计与分析系统

System Design Document

学　　号： 24325155

姓　　名： 梁桂铭

专　　业： 计算机科学与技术

完成日期： 2025 年 12 月 27 日

2025-2026 学年第一学期

目录

1 背景假设与外部依赖

1.1 项目背景

随着互联网信息的爆炸式增长，从海量文本数据中实时发现热点、监控舆情、洞察趋势变得至关重要。本项目旨在设计并实现一个基于滑动窗口的热词统计与分析系统，用以处理持续到来的文本数据流，实时维护一个时间窗口内的词频统计，并提供可配置的 Top-K 查询功能。系统重点考察流式数据处理、核心数据结构设计及性能调优能力。

1.2 系统运行假设

- **数据源格式**: 系统假设输入数据为一个文本文件，模拟流式数据。文件中的每一行代表一个事件，格式为 [H:MM:SS] 文本内容。
- **数据时序**: 假设输入文件中的数据是按照时间戳顺序排列的。系统当前设计未考虑迟到或乱序数据的处理。
- **查询指令**: 系统能识别并处理特定的查询指令（该指令是写在 input.txt 输入文件内的），格式为 [ACTION] QUERY K=N，该指令会触发一次 Top-K 结果的输出。
- **运行环境**: 系统由 C++ 后端和 Python 前端（$Python \geq 3.9$）构成，后端负责核心计算，前端负责交互展示。两者通过文件系统和 UDP 协议进行通信。

1.3 外部依赖

- **C++ 后端**:
  - **cppjieba**: 一个性能优异的 C++ 中文分词库，用于对文本进行分词处理。本项目利用其进行中文句子的切分，并支持加载自定义词典和停用词表。
- **Python 前端**:
  - **Streamlit**: 一个开源的 Python 库，用于快速创建和共享美观的数据科学和机器学习 Web 应用。本项目用其构建用户交互界面。
  - **Pandas**: 提供了高性能、易于使用的数据结构和数据分析工具，用于在前端处理和组织从后端接收到的数据。
  - **Altair**: 一个用于 Python 的声明式统计可视化库，用于将 Top-K 数据动态地绘制成交互式条形图。
- **通信协议**:

– **UDP (User Datagram Protocol)**: 后端计算引擎通过 UDP 将实时的 Top-K 统计结果以 JSON 格式发送给前端界面，以实现低延迟的数据更新。

2 模块/架构图与数据流

2.1 系统架构

本系统采用前后端分离的客户端/服务器（C/S）架构。

- **前端 (Client)**: 基于 Python 和 Streamlit 构建的 Web 用户界面。它负责接收用户上传的数据文件、配置运行参数（如窗口步长、Top-K 值、分词模式等），并动态、实时地可视化后端发送的统计结果。
- **后端 (Server)**: 一个由 C++ 编写的高性能命令行程序。它作为核心处理引擎，负责数据的读取、预处理、核心统计计算，并通过 UDP 协议将结果推送给前端。

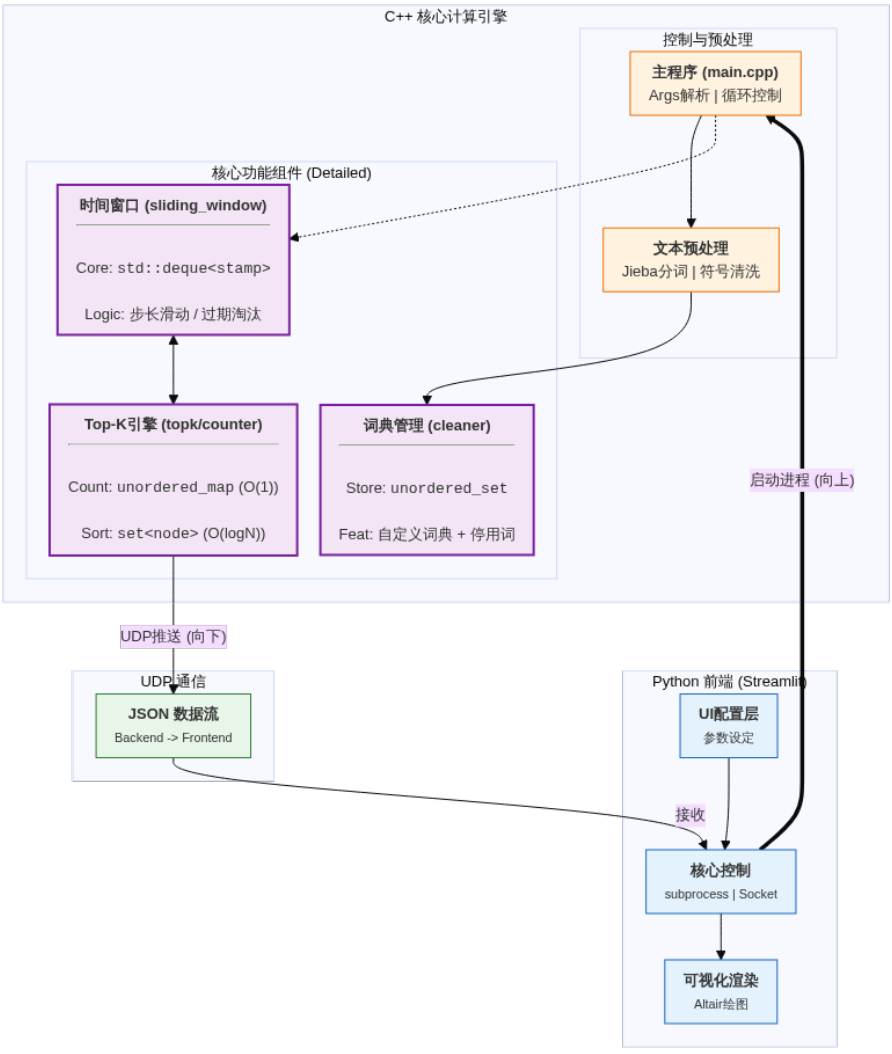

图 1: 系统架构与数据流图

2.2 核心模块划分

1. **数据采集与配置层 (前端)**: Streamlit 界面提供文件上传和参数输入功能，将用户在前端输入的参数通过 subprocess.Popen(cmd, ...) 转换为命令行指令，将用户输入的自定义停用词和自定义专用词在 temp/ 中生成临时文件。

2. **预处理层 (后端)**:

   - parserLine 函数: 解析输入行，分离时间戳和文本。
   - clean.cpp/segmentation 函数: 调用 cppjieba 库对文本进行分词。
   - clean.cpp/cleaner 函数: 加载停用词表，过滤分词结果中的无意义词汇和标点符号。

3. **统计核心层 (后端)**:

   - slide_win.cpp: 核心逻辑所在，管理一个基于时间的滑动窗口。
   - counter.cpp: 维护窗口内所有词汇的实时词频计数。
   - topk.cpp: 高效维护词频的 Top-K 列表。

4. **查询与服务层 (前后端协同)**:

   - main.cpp (后端): 监听文件中的 [ACTION] 指令，并将结果写入输出文件；接收命令行参数，读取输入文件，并将结果送入前端可视化。
   - config.h/UdpSender 类 (后端): 封装底层的 Socket 通信逻辑（支持跨平台），提供统一的 sendData 接口，负责将数据包通过 UDP 协议低延迟地推送到前端。
   - json_sender.cpp (后端): 负责业务数据的格式化，将每个时间点的 Top-K 结果序列化为 JSON 字符串，并调用 Sender 发送。
   - app.py (前端): 监听 UDP 端口，接收 JSON 数据并驱动可视化图表更新。

2.3 数据流

1. **启动**: 用户在 Streamlit 前端上传输入文件（如 input.txt）并设置步长、K 值等参数，点击 “开始分析”。

2. **进程创建**: 前端应用保存上传文件，并使用用户配置的参数作为命令行参数，启动后端的 C++ 可执行程序。

3. **数据处理循环 (后端)**:

   (a) 后端逐行读取输入文件。

(b) 对每一行文本，依次执行 **解析 -> 分词 -> 清洗**的预处理流程。

(c) 将处理后的词汇列表连同时间戳送入 **滑动窗口管理器**。

(d) 窗口管理器根据新数据的时间戳，淘汰窗口前端的过期数据。

(e) 数据的进入和淘汰都会触发 **实时计数器**和 **Top-K 维护结构**的更新。

(f) 处理完每一行后，后端将当前的 Top-K 结果打包成 JSON 字符串。

4. **实时通信**: 后端通过 UDP 将 JSON 数据包发送到本地的特定端口（例如 9999）。

5. **前端渲染**: Streamlit 前端持续监听该 UDP 端口。一旦收到数据，立即解析 JSON，并更新页面上的词频条形图，向用户展示最新的热词情况。

6. **指令响应**: 如果后端在文件中读到 [ACTION] QUERY K=N 行，它会根据指定的 K 值，从 Top-K 结构中查询结果，并将其格式化后写入指定的输出文件（如 output.txt）。

7. **结束**: 后端处理完所有输入行后，发送一个"EOF" 结束信号给前端，然后正常退出。前端收到信号后停止监听。

8. **结果展示**

(a) 在 data/文件夹中生成一个 output.txt 文件，该文件的统计结果基于 input.txt 文件中的 [ACTION]，即只统计出现该指令时的 Top-K;

(b) 在 streamlit 生成的 Web 前端页面展示的是每一个时间戳对应的 Top-K, 可自行调节时间戳和 K 的值;

3 核心数据结构设计与复杂度分析

3.1 实时计数器: unordered_map

- **数据结构**: unordered_map<string, long long> words_count
- **设计说明**: 采用哈希表（Hash Map）来存储窗口内每个独立词汇及其对应的出现次数。Key 为词汇（string），Value 为词频（long long）。
- **复杂度分析**:
  - **更新/查询 (增/删/改/查)**: 平均时间复杂度为 O(1)。最坏情况下为 O(U)，其中 U 是窗口内独立词汇的数量，但在良好的哈希函数下极为罕见。
  - **空间复杂度**: O(U)，与窗口内独立词汇数成正比。

- **设计取舍**: 相较于 map（红黑树），unordered_map 提供了更快的平均查找和更新速度。对于词频统计这种无需排序、只关心快速更新的场景，哈希表是最佳选择。
- **核心代码实现**: 以下展示的是滑动窗口在运作时 words_count 的变化，主要是删除队首元素和增加队尾元素。

```
// 当滑动窗口满容量时删除队列首的元素
void del_count(vector<string> words) {
    for (lli i = 0; i < words.size(); i++) {
        words_count[words[i]]--;
        topk.update(words[i]);
        if (words_count[words[i]] == 0) {
            // 当某个键的值清零时删除该元素
            words_count.erase(words[i]);
        }
    }
}

// 读取到新的一行时更新计数器
void add_count(vector<string> words) {
    for (lli i = 0; i < words.size(); i++) {
        words_count[words[i]]++;
        topk.update(words[i]);
    }
}

```

3.2 时间窗口管理器: deque

- **数据结构**: deque<stamp> windows，其中 stamp 是一个包含时间戳和词汇列表的结构体。
- **设计说明**: 使用双端队列（Deque）来模拟滑动窗口。当新的数据条目到来时，将其从队尾（back）压入；当窗口滑动，需要淘汰旧数据时，从队首（front）弹出。
- **复杂度分析**:
  - **入队 (push_back)**: 摊还时间复杂度为 O(1)。
  - **出队 (pop_front)**: 时间复杂度为 O(1)。
  - **空间复杂度**: O(N * L)，其中 N 是窗口内包含的数据行数，L 是每行的平均词汇数。
- **设计取舍**: deque 完美契合了滑动窗口“先进先出”的特性，其在两端进行增删操作的效率远高于 vector（其头部删除是 O(N) 的）。

- **核心代码实现**: 时间窗口管理器的核心代码，包括在第一行文本时对时间窗口进行初始化，在遍历文本的过程中对时间窗口实时更新。其中 front_of_deque 用于记录当前时间管理器的最前端时间戳，back_of_deque 用于记录最后端的时间戳。

```
// 这里的stride以秒为单位
// 初始化滑动窗口
void init_win(EventToken e, const string& motion) {
    vector<string> seg = segmentation(e.words, motion);
    stamp new_st = {e.timestamp, seg};
    add_count(seg);
    windows.push_back(new_st);
    front_of_deque = e.timestamp;
    back_of_deque = e.timestamp;
}
// 对滑动窗口进行实时更新
void update_win(EventToken e, lli stride, const string& motion) {
    while (time_sub(front_of_deque, e.timestamp) > stride) {
        del_count(windows.front().words);
        windows.pop_front();
        if (!windows.empty()) {
            front_of_deque = windows.front().timestamp;
        }
        else break;
    }
    vector<string> seg = segmentation(e.words, motion);
    if (windows.empty()) {
        front_of_deque = e.timestamp;
    }
    back_of_deque = e.timestamp;
    stamp new_st = {e.timestamp, seg};
    windows.push_back(new_st);
    add_count(seg);
}

```

3.3 Top-K 维护结构: set + unordered_map

- **数据结构**:

1. set<node, greater<node> > S: 一个基于红黑树的集合,用于存储节点 nodeword, count。集合根据词频 count 进行降序排序。
2. unordered_map<string, set<...>::iterator> pos: 一个哈希表，将词汇映射到其在 set 中的迭代器。

- **设计说明**: 这是一个经典且高效的动态 Top-K 解决方案。
  - set 自动维护了所有词汇按词频排序的顺序，获取 Top-K 结果只需遍历其前 K 个元素。
  - 当一个词的词频更新时，我们不能直接修改 set 中的元素。正确做法是：先通过 pos 哈希表以 O(1) 的平均时间找到该词的旧节点迭代器，然后在 set 中删除旧节点，最后插入包含新词频的新节点。
- **复杂度分析**:
  - **更新词频 (Update)**: 复杂度为 O(log U)。此操作包含一次哈希查找 (平均 O(1))，一次 set 删除 (O(log U)) 和一次 set 插入 (O(log U))。
  - **获取 Top-K (getTopK)**: 复杂度为 O(K)，只需遍历 set 的前 K 个元素。
  - **空间复杂度**: O(U)，因为两个结构都需要存储所有独立词汇的信息。
- **设计取舍**: 相比于每次更新后对所有词汇进行排序（O(U log U)），或使用大小为 K 的小顶堆（更新非 Top-K 词汇时成本高），本方案在频繁更新和查询的场景下提供了优秀的综合性能。它保证了每次更新的对数时间复杂度，同时能即时提供有序的 Top-K 列表。
- **核心代码实现**: 包括对 set 的更新和当前 Top-K 元素的获取。

```
// 更新Top-K set
void TopK::update(const string& w) {
    // 删除旧节点 (如果有)
    if (pos.find(w) != pos.end()) {
        auto it = pos[w];
        // it是否超出S的范围, 查看该元素是否在S中
        if (it != S.end() && S.find(*it) != S.end()) {  // 确保迭代器有效
            S.erase(it);
        } else {
            cerr << "[WARN] Iterator invalid or element not found in S for word: " << w << endl;
        }
        pos.erase(w);  // 从 pos 中移除无效的迭代器
    }

    // 插入新节点
    if (words_count[w] > 0) {
        auto newi = S.insert({w, words_count[w]}).first;
        pos[w] = newi;
    }
```

```
}

// 获取当前Top-K
vector<pair<string, lli>> TopK::getTopK() {
    vector<pair<string, lli>> result;
    auto it = S.begin();
    for (lli i = 0; i < K && it != S.end();i++, it++) {
        result.push_back({it -> word, it -> cnt});
    }
    return result;
}

```

4 滑动窗口定义与实时性保证

4.1 滑动窗口定义

本系统采用 **基于时间的滑动窗口**。

- **窗口大小**: 由一个可配置的时间跨度（stride，单位：秒）定义。在任意时刻，窗口包含的数据范围是从当前最新进入的数据行的时间戳 T_latest 向前回溯 stride 秒。即，窗口覆盖的时间区间为 [T_latest - stride, T_latest]。
- **滑动机制**: 窗口的滑动是由新数据的到达驱动的。当一个时间戳为 T_new 的新数据行被处理时，系统会检查并淘汰掉窗口中所有时间戳 T_old 满足 T_new - T_old > stride 的数据。这个淘汰过程是从窗口最前端（即时间最早的数据）开始的。

4.2 实时性保证

系统的实时性主要通过以下几点来保证：

1. **高效的数据结构**: 如上一节所述，所有核心计算（词频统计、窗口维护、Top-K 更新）均采用了高时间效率的数据结构，确保单条数据的处理延迟极低（主要在 O(log U) 级别）。
2. **流式处理模型**: 系统以后端读取文件的方式模拟数据流，每处理完一行数据就立即更新内部状态并向前端发送结果，而不是等待整个文件处理完毕。
3. **低延迟通信**: 后端与前端之间采用 UDP 协议进行通信。UDP 是一种无连接的协议，开销小，传输延迟低，非常适合这种需要频繁发送小数据包的实时更新场景。

5 性能优化与资源评估方法

5.1 性能优化措施

- **算法与数据结构层面**: 这是本系统最核心的优化。通过采用哈希表、双端队列以及“红黑树 + 哈希表”的组合，将关键操作的时间复杂度降至常数或对数级别。
- **I/O 优化**: 在 C++ 后端，使用 ios::binary 模式读取文件，可以避免不同操作系统（Windows CRLF vs. Linux LF）换行符处理的潜在问题和开销。
- **字符串处理**: 在需要拼接字符串的场景(例如 json_sender.cpp),优先使用 stringstream 或类似机制，相比于简单的字符串重复 + 操作，可以减少内存的重复分配和拷贝，提升效率。
- **依赖库选择**: 选用高性能的 cppjieba 作为分词引擎，其本身经过了良好的优化，保证了文本预处理阶段的效率。

5.2 资源评估方法

对系统性能和资源的评估，应从时间、空间和吞吐量等维度进行。

- **时间复杂度评估**:
  - **单行处理**: 处理一行包含 L 个词的文本，时间复杂度主要由 Top-K 更新决定，为 O(L * log U)，其中 U 是窗口内独立词总数。
  - **整体处理**: 处理一个包含 M 行的文件，总时间复杂度约为 O(M * L_avg * log U_avg)。
- **内存占用评估**:
  - 内存主要消耗在核心数据结构上:
    1. words_count 哈希表: O(U)
    2. windows 双端队列: O(N * L) (窗口内总词数)
    3. Top-K 维护结构 (set + map): O(U)
    4. cppjieba 加载的词典。
  - 内存占用的峰值与窗口大小（stride）和该时间窗口内数据的复杂度（独立词汇量 U）正相关。
- **测试方法**: 在性能测试报告中，设计不同负载下的测试用例，记录并绘制系统在不同负载下的吞吐量、延迟和内存占用图表，以全面评估系统性能。

6 引用规范与开源依赖

本项目严格遵守学术诚信与开源引用规范。所有使用的第三方代码库均已在此明确列出，未引用任何未注明来源的第三方代码。系统开发中调用的开源库及其版本、许可证信息如下表所示：

表 1: 开源依赖库清单

组件名称	应用模块	版本 (Version)	许可证 (License)
cppjieba	C++ 后端 (核心分词)	v5.6.0	MIT License
Streamlit	Python 前端 (Web UI)	v1.52.2	Apache-2.0
Pandas	Python 前端 (数据处理)	v2.3.3	BSD-3-Clause
Altair	Python 前端 (可视化)	v6.0.0	Vega-Altair Developers

6.1 依赖库详情与来源说明

1. **cppjieba**
   - **来源**: https://github.com/yanyiwu/cppjieba
   - **用途**: 本项目后端的核心依赖。这是一个 Header-only 的 C++ 中文分词库，本项目复用了其 HMM 模型和字典加载逻辑，用于对输入文本流进行实时分词。
   - **许可证说明**: MIT 许可证允许在保留版权声明的前提下进行修改和分发，本项目符合该规定。
2. **Streamlit**
   - **来源**: https://github.com/streamlit/streamlit
   - **用途**: 用于构建前端交互界面，提供了文件上传组件、参数配置侧边栏以及图表容器。
3. **Pandas**
   - **来源**: https://github.com/pandas-dev/pandas
   - **用途**: 在前端接收到 UDP JSON 数据后，使用 DataFrame 结构对 Top-K 词频数据进行结构化存储和排序，以便于后续渲染。
4. **Altair**

- **来源**: https://github.com/altair-viz/altair
- **用途**: 基于 Vega-Lite 的声明式统计可视化库，用于在 Streamlit 中绘制高性能的实时词频条形图。