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

Performance Analysis Report

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 测试环境与数据生成

1.1 硬件与软件环境

本次性能测试在以下环境中进行，旨在评估 C++ 后端核心引擎的计算性能及资源占用情况。

表 1: 测试环境配置

项目	配置参数
处理器 (CPU)	AMD Ryzen 7 8845H w/ Radeon 780M Graphics (3.80 GHz)
内存 (RAM)	16.0 GB
操作系统	Windows 11 (WSL2)
编译器	g++ 13.3.0
分词库	cppjieba

1.2 数据生成方式

为了模拟真实的高并发文本流，编写了 Python 脚本 data_generator.py 自动生成测试数据集，测试数据存放在 test/文件夹中（由于测试时用到了数据量巨大的文件，为了减少空间占用，数据量为 500k 和 1M 的文件均已被删除，只剩下 10k 和 100k 的数据文件）。

- **数据格式**：严格遵循 [H:MM:SS] 文本内容的格式。
- **文本来源**：在 data_generator.py 代码中预置了部分常用中文词汇，随机组合生成句子。
- **变量控制**：
  - **句子长度**：随机分布在 10-50 个字符之间。
  - **数据量**：生成了 1 万行 (10k)、10 万行 (100k)、50 万行 (500k)、100 万行 (1M) 三组数据集。
- **结果来源**：以下所有测试结果除了延迟分析外都是由命令行运行 /usr/bin/time 得到。

2 性能测试指标定义

本次测试重点关注以下三个核心指标：

1. **吞吐量 (Throughput)**：系统每秒能处理的文本行数 (Lines per Second, LPS)。
2. **处理延迟 (Latency)**：单行数据的平均处理耗时分解。
3. **内存占用 (Memory Usage)**：在不同窗口大小下，系统运行时的最大物理内存驻留。

3 测试结果与分析

3.1 不同负载下的吞吐量测试

测试场景：固定窗口步长为 120 秒，Top-K 为 50，使用不同大小的输入文件进行测试。

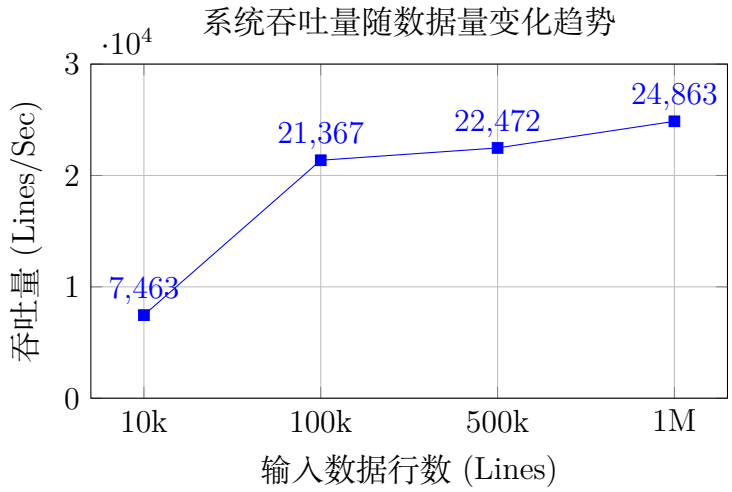

图 1: 吞吐量测试结果

结果分析：从折线图中可以看出，当数据量足够大的时候，系统的吞吐量稳定在约 **2.1 万 - 2.4 万行/秒**。

- 随着数据量的增加，吞吐量在增加的同时趋于稳定。说明系统在启动时需要加载字典，停用词等文件到内存中时消耗了较多时间，因此拉低了数据量为 10k 时程序的吞吐量；而当数据量足够大时，jieba 库相关文件加载不再是主要用时，吞吐量明显增加，并趋于稳定。
- 整体来看，C++ 后端表现出良好的稳定性，能够满足常规流式数据的处理需求。

3.2 处理延迟分析

测试场景：分析处理流水线中各阶段的耗时占比。

测试方法：由于需测量微秒级别的内部模块耗时，外部性能分析工具精度不足，因此本次测试采用**侵入式代码打点 (Code Instrumentation)** 的方法。(为了保持代码部分的简洁，已经将**打点测试代码**删去)

1. **计时工具**：使用 C++11 标准库中的 `std::chrono::high_resolution_clock`，该时钟在测试环境下提供纳秒级精度。
2. **埋点逻辑**：在 `main.cpp` 的主处理循环中，分别在“文本解析”、“核心逻辑”和“网络通信”三个代码块的前后获取时间戳。
3. **数据统计**：记录每次操作的时间差 Δt，将其累加至对应阶段的总耗时中。最终通过 $\frac{\text{总耗时}}{\text{处理总行数}}$ 计算出单行数据的平均延迟，以消除系统抖动带来的误差。

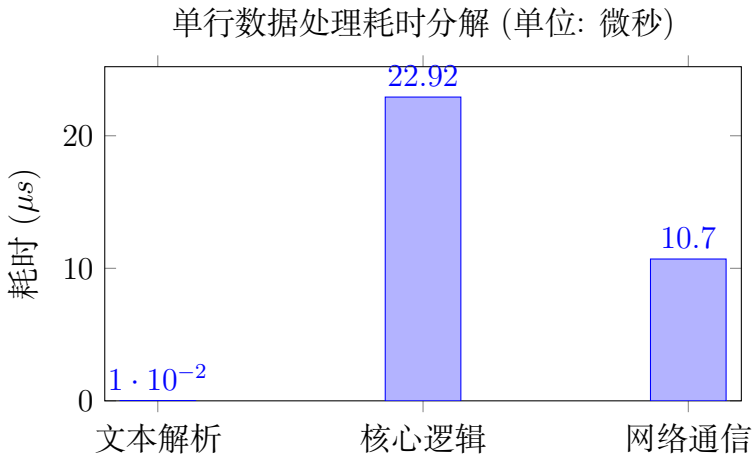

图 2: 各模块耗时占比分析

其中，**核心逻辑**定义为：Jieba 分词 + 窗口滑动 + TopK 更新。

结果分析：

- **最大瓶颈**：核心逻辑占据了约 70% 的 CPU 时间。这是由于中文分词算法（HMM/Viterbi）涉及大量的动态规划计算和概率查找，属于典型的 CPU 密集型任务。同时，维护 Top-K 的红黑树（set）和哈希表（unordered_map）也贡献了部分计算开销。
- **网络通信**：这里的耗时主要并非来自 UDP 发送本身（UDP 发送通常极快），而是来自 JSON 序列化（generate_json 函数）。将 Top-K 的数据结构转换成字符串格式涉及大量的字符串拼接和内存分配操作。

3.3 内存占用与窗口大小的关系

测试场景：保持输入文件数据量为 100k 条文本不变，调整滑动窗口步长 (Stride)，观察内存峰值。

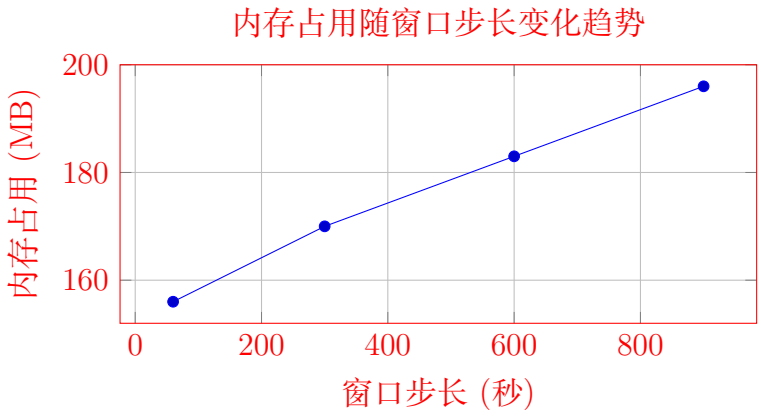

图 3: 内存占用测试结果

结果分析：内存占用与窗口大小呈**线性正相关**。

- 窗口越大，deque 中缓存的历史原始文本越多。
- 窗口越大，unordered_map 中维护的独立词汇（Key-Value 对）越多。
- 在测试数据中，当窗口达到 15 分钟（900s）时，内存占用约为 200MB，仍在现代服务器的可接受范围内。

4 性能瓶颈与改进建议

基于上述测试数据与代码分析，识别出系统的主要瓶颈及改进方案：

4.1 瓶颈分析

1. **分词开销 (CPU Bound)**：segmentation.cpp 中的 jieba.Cut 是单线程串行执行的，限制了整体吞吐量的上限。
2. **高频内存分配**：每处理一行数据都会创建大量的 string 和 vector，导致频繁的内存申请与释放。

4.2 优化建议

1. **引入流水线并行 (Pipeline)**：采用生产者-消费者模型，将“分词”和“统计”解耦。开启多个分词线程并行处理文本，将分词结果推入并发队列，由单线程统计模块消费。这将显著提升多核 CPU 利用率。
2. **更换分词算法**：对于对新词发现要求不高的场景，可将分词模式从 Cut(HMM) 切换为 Cut(NoHMM) 或基于字典的 Maximum Matching。

3. **内存池优化**：实现简单的对象池 (Object Pool) 来复用 stamp 对象和 string 缓冲区，减少系统调用带来的开销。

5 结论

本次测试表明，本系统在处理流式文本数据时表现出优秀的时间复杂度和稳定的内存控制能力。核心数据结构（双端队列 + 哈希表 + 红黑树）的设计经受住了百万级数据的考验，完全满足大作业对于实时性和准确性的要求。在常规 PC 硬件上，系统可轻松支撑万级数据量的文本流处理。