Computer Science and Application 计算机科学与应用, 2021, 11(5), 1512-1521
Published Online May 2021 in Hans. http://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2021.115155

Hans汉斯

一种基于动态水位值的Flink调度优化算法

冯　鹏[1,2]，黄　山[2,3,4]，段晓东[2,3,4]

[1]大连大学信息工程学院，辽宁　大连
[2]大数据应用技术国家民委重点实验室，辽宁　大连
[3]大连民族大学计算机科学与工程学院，辽宁　大连
[4]大连市民族文化数字技术重点实验室，辽宁　大连
Email: 2324290252@qq.com, huangshan@dlnu.edu.cn, duanxd@dlnu.edu.cn

收稿日期：2021年4月25日；录用日期：2021年5月20日；发布日期：2021年5月27日

摘　要

新一代大数据间引擎Flink在面临远程传输问题时，主要通过Netty完成数据传输，并依靠Netty水位值机制来保证其反压机制的运行。Netty水位值机制是一种相对静态的机制，这使得Flink在面临突变性特别大的数据流时会反复进行反压，进而影响整个Flink集群的计算效率。针对此问题，本文提出一种基于动态水位值的Flink调度优化算法Flink-N，经实验验证，与Flink默认的反压机制相比，Flink-N在吐吞量、CPU利用率及时延均有很大提升，时延整体优化达18%，最高优化23%。

关键词

Flink，大数据，反压，Netty

A Flink Scheduling Optimization Algorithm Based on Dynamic Water Level

Peng Feng[1,2], Shan Huang[2,3,4], Xiaodong Duan[2,3,4]

[1]College of Information, Dalian University, Dalian Liaoning
[2]State Ethnic Affairs Commission Key Laboratory of Big Data Applied Technology, Dalian Liaoning
[3]College of Computer Science and Technology, Dalian Minzu University, Dalian Liaoning
[4]Dalian Key Laboratory of Digital Technology for National Culture, Dalian Liaoning
Email: 2324290252@qq.com, huangshan@dlnu.edu.cn, duanxd@dlnu.edu.cn

Received: Apr. 25th, 2021; accepted: May 20th, 2021; published: May 27th, 2021

文章引用: 冯鹏, 黄山, 段晓东. 一种基于动态水位值的 Flink 调度优化算法[J]. 计算机科学与应用, 2021, 11(5): 1512-1521. DOI: 10.12677/csa.2021.115155

Abstract

When the new generation of big data engine Flink is faced with the problem of remote transmission, it mainly completes the data transmission through Netty, and relies on the Netty water level mechanism to ensure the operation of its back pressure mechanism. Netty water level mechanism is a relatively static mechanism, which makes Flink repeatedly back pressure in the face of catastrophic data flow, thus affecting the computing efficiency of the whole Flink cluster. To solve this problem, this paper proposes a Flink scheduling optimization algorithm Flink-N based on dynamic water level. Experimental results show that, compared with Flink's default back pressure mechanism, Flink-N greatly improves the throughput, CPU utilization and time delay. The overall delay optimization is 18%, and the maximum optimization is 23%.

Keywords

Flink, Big Data, Backpressure, Netty

1. 引言

随着信息时代的不断发展及线上线下交互需求的增加，实时计算能力强的流处理系统越来越被产业办重视。Flink 因其具有强大的实时计算能力被广泛应用于实时计算[1] [2]、高频交易和社交网络等方面。如美团[3]、阿里云实时计算[4]及在线监测[5]等场景。

反压(backpressure)能力的好坏是判断实时处理系统数据处理能力强弱的重要依据。当系统瞬时负载高峰使得其接受数据的速度远远高于其处理数据的能力时，就会出现反压情况，如垃圾回收不及时或者停顿可能使得流入系统的数据快速堆积、大促或秒杀活动时出现的流量陡增等都会造成反压。如不对反压及时处理，将会使系统资源耗尽甚至导致系统崩溃。

Flink 通过自身数据流来响应反压问题，下游的消费者处理数据减慢会降低上游发送者的发送速率。

2. 相关研究现状

现有大数据实时处理系统处理反压问题方面，Storm [6] [7]是通过监控 Bolt 中的接收队列负载情况，如果超过高水位值就会将反压信息写到 Zookeeper，Zookeeper 上的 watch 会通知该拓扑的所有 Worker 都进入反压状态，最后 Spout 停止发送 tuple。J Storm [8] [9]采用逐级降速的方式来处理反压，使用 Topology Master 替代 Zookeeper 来协调拓扑进入反压状态，效果较 Storm 更为稳定。Spark Streaming [10] [11]根据批处理时间(Batch Processing Time)和批次间隔(Batch Interval，即 Batch Duration)的信息来动态调整系统的摄入速率，从而完成其反压工作。

在 Flink 优化方面，关沫使用 Flink 执行一种传统堆序优化后的算法 Heap Optimize，增加了 Flink 的吞吐量[12]。针对 Flink 默认的先来先服务的任务调度策略，王丽娟等人[13]通过资源感知，将待执行任务分配到最佳节点进行计算，优化了 Flink 的负载均衡。何贞贞等人[14]则根据任务间数据流的大小确定拓扑边的权重，以生成关键路径，大幅缩减了 Flink 节点间的通信开销。文献[15] [16] [17]把 Flink 从原

来的 CPU 迁移扩展到异构的 CPU-GPU 集群，在并行计算、内存管理及通信策略方面极大地提高了 Flink 的计算能力。

目前诸多的研究当中没有 Flink 反压方面的问题，当 Flink 面临远端传输问题时，其所依托的 Netty 所采用的是一种静态的水位机制，这使得 Flink 在面临颠簸状态数据的远程传输问题时，容易出现反复反压的情况，极大地影响了 Flink 传输数据的效率，故而本文将针对此问题展开研究。

3. Flink 反压原理解析

3.1. Flink 反压原理

Flink 的反压原理如图 1 所示，假如 Flink 的一个 Job 分为 Task A、B、C，其中 Task A 是 Source Task、Task B 处理数据、Task C 为 Sink Task。假如 Task C 由于各种原因吞吐量降低，会将负载信息反馈给 Task B，Task B 会降低向 Task C 发送数据的速率，此时若 Task B 还保持从 Task A 读取数据，数据会把 Task B 的 Send Buffer 和 Receive Buffer 撑爆，导致 OOM 或者丢失数据。所以，当 Task B 的 Send Buffer 和 Receive Buffer 被用完后，Task B 会用同样的原理将负载信息反馈给 Task A，Task A 收到 Task B 的负载信息后，会降低给 Task B 发送数据的速率，以此类推。

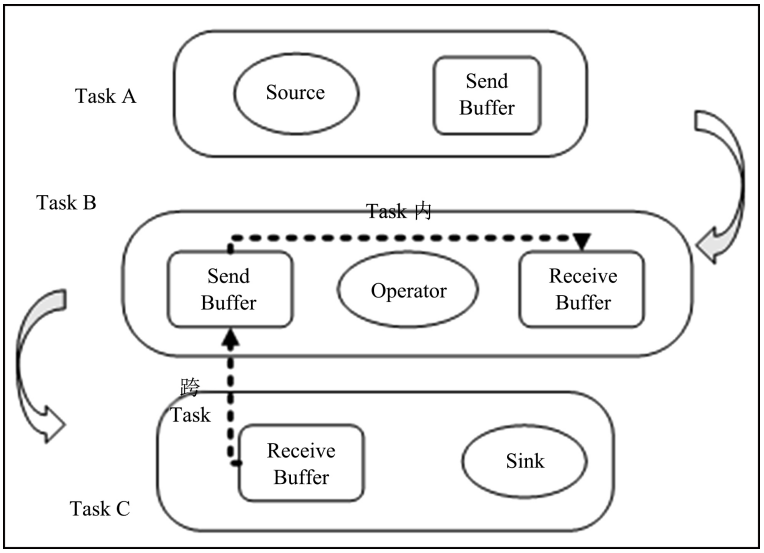

Figure 1. Flink back pressure schematic diagram
图 1. Flink 反压原理图

3.2. Flink 网络传输的数据流向

Flink 反压存在 Task 内与跨 Task 两种情况，本文已在图 1 中标注，本文主要是针对 Flink 跨 Task 传输进行反压优化，故下文主要对 Flink 跨 Task 传输进行介绍：图 2 展示了 Flink 网络传输时的数据流向，可以看到 Task Manager A 给 TaskManager B 发送数据，Task Manager A 做为 Producer，Task Manager B 做为 Consumer。Producer 端的 Operator 实例会产生数据，最后通过网络发送给 Consumer 端的 Operator 实例。Producer 端 Operator 实例生产的数据首先缓存到 Task Manager 内部的 Net Work Buffer。Net Work 依赖 Netty 来做通信，Producer 端的 Netty 内部有 Channel Outbound Buffer，Consumer 端的 Netty 内部有 Channel Inbound Buffer。Netty 最终还是要通过 Socket 发送网络请求，Socket 这一层也会有 Buffer，Producer 端有 Send Buffer，Consumer 端有 Receive Buffer。

故 Flink 网络传输时的整个反压过程为：首先 Producer Operator 从自己的上游或者外部数据源读取到数据后，对一条条的数据进行处理，处理完的数据首先输出到 Producer Operator 对应的 Net Work Buffer 中。Buffer 写满或者超时后，就会触发将 Net Work Buffer 中的数据拷贝到 Producer 端 Netty 的 Channel Outbound Buffer，之后又把数据拷贝到 Socket 的 Send Buffer 中，这里有一个从用户态拷贝到内核态的过程，最后通过 Socket 发送网络请求，把 Send Buffer 中的数据发送到 Consumer 端的 Receive Buffer。数据到达 Consumer 端后，再依次从 Socket 的 Receive Buffer 拷贝到 Netty 的 Channel Inbound Buffer，再拷贝到 Consumer Operator 的 Net Work Buffer，最后 Consumer Operator 就可以读到数据进行处理了，这就是两个 Task Manager 之间的数据传输过程。

4. Flink 反压机制的缺点

如图 2 所示，Flink 通过 Netty 完成其数据的网络传输任务，Netty 在向底层的 Channel 写数据的时候会用到 Channel Outbound Buffer，Channel Outbound Buffer 本身是无界的，如果水位控制不当的话就会造成占用大量的内存，因此 Netty 为其配置了一个高水位线和低水位线。为避免上游数据量太大，当上游数据的大小超过高水位线的时候对应 channel 的 isWritable 就会变成 false，当上游数据的大小低于低水位线的时候，isWritable 就会变成 true，低水位线主要是其自动恢复运行的一种保障，为便于理解下文所提的水位线均指高水位线。Flink 以此来保证不在网络中写入太多数据，进而保证 Flink 的反压能力。

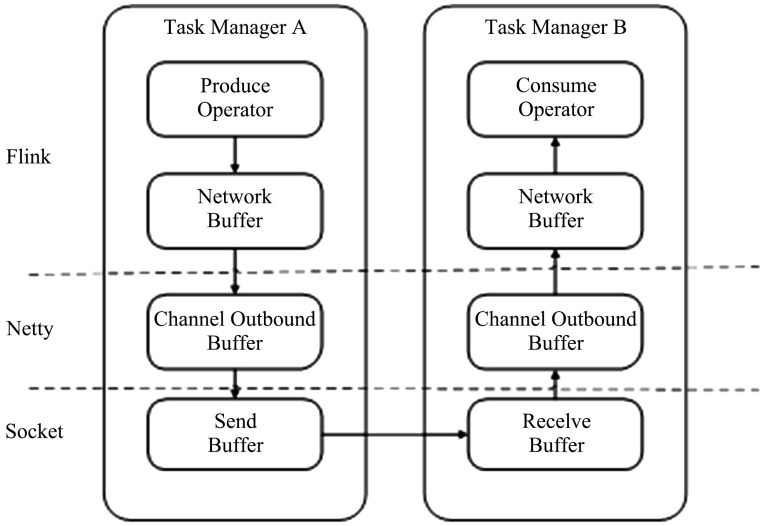

Figure 2. Data flow chart of Flink network transmission
图 2. Flink 网络传输数据流向图

分析源码可知，Netty 水位机制是一种静态的机制，Netty 默认其水位线的高度为定值，这使得 Flink 系统在面临瞬时流量不稳定的场景(即系统的数据流量值在特别高与特别低的值之间不断跳动时)时，会出现下述两种问题：

1) 水位值较下游可用缓存区数偏低：如图 3 (左图)所示，图中以“圆圈”表示数据，以“方框”表示缓存区的大小，下同。假设当上游 A 点来临的数据量是 9 (Flink 中以 buffer 为数据单位，每个 buffer 大小为 32 k，为便于表述，下文块描述)，而此时下游 B 点的可用缓存区是 10，H 代表代表数据通道(其作用类似于水坝，水位值的大小决定了其单位时间通过的数据量大小)，此处设水位值高度为 4，则 Flink 传输本批次的数据需要 3 个单位时间(上游共 9 块数据，每个单位时间只能通过 4 块的数据，需要 3 个单

位的时间来处理这批数据)。而若此时的水位值为 9 或者 10 的话，则只需要一个单位时间，Flink 便可以处理本批次的数据。

2) 水位值较下游可用缓存区数偏高：如图 3 (右图)，假设当上游 A 点来临的数据量是 4，而此时下游 B 点的可用缓存区为 2，水位值高度为 4。由于数据量不于水位值高度，Flink 会误以为可以在一单位时间内接受这批数据，如图中可以看出，只有 2 块的缓存区，直接接收了 4 块的数据量，会直接导致内存溢出(OOM)甚至引起系统阻塞。综上，由于不合理的静态水位线的设置，使得 Flink 传输数据时间延长，或者出现非正常的阻塞，进而影响整个 Flink 的数据传输情况。

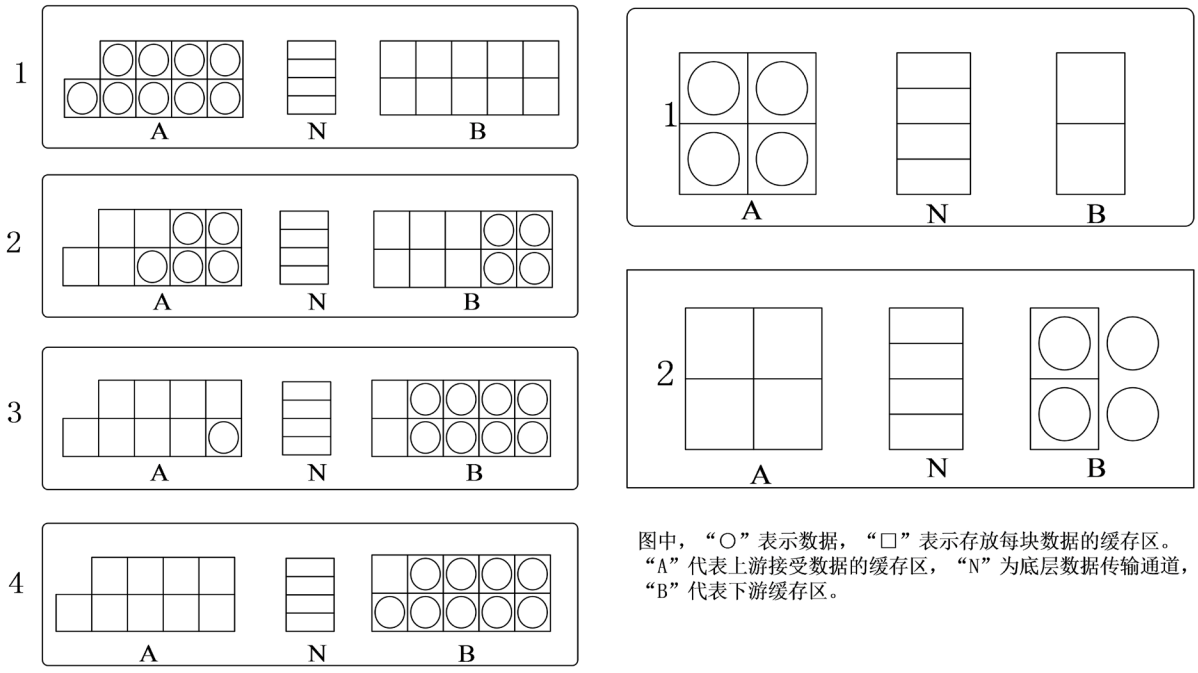

Figure 3. Schematic diagram of data transmission of different Netty water level values
图 3. 不同 Netty 水位值数据传输示意图

综上，由于不合理的静态水位线的设置，使得 Flink 传输数据时间延长，或者出现非正常的阻塞，进而影响整个 Flink 的数据传输情况。

5. 反压优化算法

本节将针对 Flink 反压传输所存在的缺点，提出一种基于动态水位值的 Flink 调度优化算法，并给出示例进行说明：

5.1. 基于动态水位值的 Flink 调度优化算法

虽然可以在数据处理前对 Netty 所默认的两个 buffer 高度进行参数调整，但这种默认的定值始终是一种静态的机制。这种相对静态的机制使得 Flink 在面临远程传输问题时，容易出现上文所述的两种问题。本节将针对 Flink 反压传输所存在的缺点，提出一种基于动态水位值 Flink 调度优化算法，并给出例子进行说明。Flink-N 算法的核心思想是：把 Flink 中 Netty 下游可用 buffer 数 B_t 实时写入 Redis 中，根据 Redis 中前后时刻 buffer 数(即 B_t 值)的大小变化，对水位值 Wt 进行动态调整，算法流程如图 4 所示，其具体步骤如下：

第一步，设置访问函数，并创建接口，使得 Flink 启动的同时运行访问函数。其中，访问函数的作用是，每间隔一段时间访问 Netty 下游缓存区可用 buffer (图中 B 点位置)的数量，并将其记录到 Redis 中；

第二步，获得下游可用 buffer 数 B_t；

第三步，取 0.8 倍的 B_0 值的整数部分(向下取整)作为 Netty 的高水位值，即令 $W_{0H} = \lfloor 0.8|B_0| \rfloor$；

第四步，将 B_t 值反馈到 Redis 中并记录；

第五步，根据 B_t 值调整水位值 W_t，具体方法为：若 B_t 大于 B_t 与 B_{t-1} 的平均值，则 W_t 取 B_t 大于 B_t 与 B_{t-1} 的平均值；反之，当 $B_t \le B_{t-1}$ 时，则令 $W_t = B_t$。

第六步，重复第二步、第四步与第五步。

本文选择 Redis 是因为 Flink 处理时延是 ms 级别的，而 Redis 数据读取速度可达 110,000 次/s，写数据的速度可达 81,000 次/s，选择 Redis 相较于其他数据库而言，不会对 Flink 的时效性产生负增益。

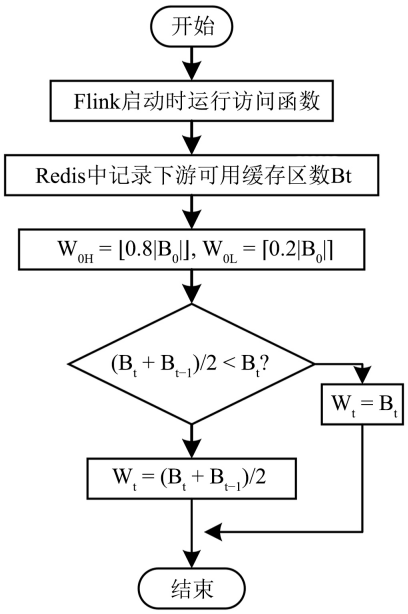

Figure 4. Flow chart of Flink-N
图 4. Flink-N 算法流程图

5.2. 示例说明

现对所提出的算法给出实例加以说明：设某连续时间段 t_1、t_2、t_3、t_4、t_5 内 Netty 上游来临的数据量分别 8、4、8、4、8，且下游对应时间段内的缓存区数分别为 10、4、10、4、10，在表 1 中以 G 表示对应时刻系统接收的数据量，以 R 表示系统实际的数据传输量。则两种不同的机制对应的数据传输结果如下(假设此处静态水位值为 6，且每次阻塞系统需要两个时刻的时间才能恢复正常运行。)：

如表 1 所示，对于 Flink 默认的反压算法，系统 t_1 时刻要传输 8 单位的数据量，默认的水位值一直是 6，故而 Flink 在 t_1 时刻只能传输 6 块的数据。在 t_2 时刻系统接收外部来的 4 块数据，同时要传输 t_1 时刻剩余的 2 块数据，因此实际传输数据为 6 块，此时水位值为 6，但下游缓存区只有 4，故而会出现第 4 节中的第二种情况而导致系统阻塞。因为发生了阻塞，系统在 t_3、t_4 会自行调整恢复到可运行状态，无法进行数据的传输。t_5、t_6、t_7、t_8 时刻的状态与 t_1、t_2、t_3、t_4 时刻的状态相似。t_{10} 时，系统已无外部的数据要接收，但仍有 t_9 时刻的 2 块数据需要传输，此时水位值为 6，下游缓存区数为 4，可直接传输，从而完成所有数据的传输。

Table 1. Comparison of data transmission examples under two algorithms
表 1. 两种算法下数据传输示例对照

时刻	数据量	缓存区数	Flink		Flink-N		传输情况
			实际传输	W_1	实际传输	W_2	
t_1	8	10	8	6	8	8	Flink 在 t_1 时剩 2 块数据；Flink-N 全部传输
t_2	4	4	6	6	4	4	Flink 发生第 1 次阻塞；Flink-N 全部传输
t_3	8	10	8	6	8	7	Flink 处于第 1 次阻塞中；Flink-N 剩余 1 块
t_4	4	4	12	6	5	4	Flink 发生第 2 次阻塞；Flink-N 剩余 1 块
t_5	8	10	14	6	9	7	Flink 处于第 2 次阻塞中；Flink-N 剩余 2 块
t_6	0	4	14	6	2	2	Flink 发生第 3 次阻塞；Flink-N 完成任务
t_7	0	10	8	6	--	--	Flink 处于第 3 次阻塞中；Flink-N 完成任务
t_8	0	4	8	6	--	--	Flink 发生第 4 次阻塞；Flink-N 完成任务
t_9	0	10	2	6	--	--	Flink 处于第 4 次阻塞中；Flink-N 完成任务
t_{10}	0	4	2	6	--	--	Flink 完成任务；Flink-N 完成任务

对于 Flink-N 算法，因为 t_1 时刻下游缓存区数为 10，所以我们为水位值赋初始值 8，此时要传输的数据为 8，故传输后无数据剩余。t_2 时刻时，下游缓存区数为 4，因为 10 与 4 的平均值 7 大于 4，所以水位值设为 4，上游数据为 4，且下游缓存区为 4，刚好可以把 t_2 时刻的数据传输完毕。t_3 时刻时，下游缓存区为 10，因为 10 与 4 的平均值 7 小于 10，所以水位值设为 7，此时上游来临的数据为 8，下游缓存区数为 10，故 t_3 时刻传输后，系统会剩余 1 单位数据在 t_4 时刻传输。t_4 时刻时，下游缓存区数为 4，因为 10 与 4 的平均值大于 4，所以 t_4 时刻的水位值设为 4，此时外部数据为 4，加之 t_3 时刻的数据剩余，整个 t_4 时刻系统实际要传输的数据为 5，此时下游缓存区为 4，因此 t_4 时刻后，系统会剩余 1 块数据在 t5 时刻传输。t_5 时刻下游缓存区数为 10，因为 10 与 4 的平均值 7 小于 10，故水位值设为 7，此时外部数据为 8，加上 t_4 时刻剩余的 1 块数据，t_5 时刻实际要传输的数据为 9，则 t_5 时刻后系统剩余 2 块的数据在 t_6 时刻传输。在 t_6 时刻时，系统下游缓存区为 4，因为 10 与 4 的平均值 7 大于 4，故水位值设为 4，此时系统外部无待接收数据，加上 t_5 时刻剩余的 2 块数据，则系统 t_6 实际要传输的数据为 2，可以全部完成传输。

综上，对于本例所给出的数据颠簸的案例，Flink 完成整个传输需要 10 个时刻，Flink-N 只需 6 个时刻即可完成数据传输。显然，理论上 Flink-N 较 Flink 默认的反压算法具有更好的时效性。

6. 实验分析

6.1. 实验环境

实验在三台配置均为 Inter(R)Core(TM) i7-8700、2.66 GHz CPU、32GDDR4 内存、2T 硬盘的机器所组成的集群上进行，采用一主两从结构，其他配置如表 2 所示：

Table 2. Experimental configuration
表 2. 实验配置

配置	参数
操作系统	CentOs7.1611
JDK	1.8.0_191
开发环境	IntelliJIDEA 2018
Flink 版本	1.4.2
Redis 版本	5.0.10

6.2. 实验及分析

实验选择典型大数据处理应用单词统计 Word Count 进行对比实验，将 Flink 默认反压机制与优化后的算法机制 Flink-N 进行对比。

实验分别从 kafka 集群中订阅同一个 Topic，而 kafka 不断读取合成的人工数据集并向该 Topic 注入数据。Flink 处理的数据便会随着时间的变化越来越多。为了消除不确定因素对实验结果的影响，本文将实验分别运行 10 组，取统计的平均值进行分析，分别观察两组实验各个时间点的吞吐量、CPU 利用率及传输时延。

1) 吞吐量效果评测

如图 5 显示的是 Flink 默认反压机制与 Flink-N 在吞吐量方面的差别，相比于默认反压机制，Flink-N 的反压机制在整体吞吐量上提高了 20%左右。

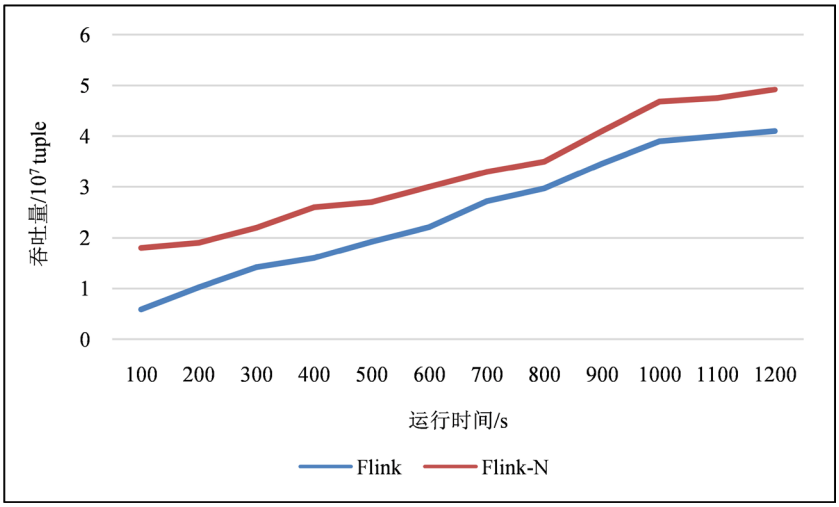

Figure 5. Throughput test comparison
图 5. 吞吐量测试对比

2) CPU 利用率效果评测

如图 6 显示的是 Flink 默认反压机制与 Flink-N 在 CPU 利用率方面的差别，相比于默认的 Flink 反压机制，Flink-N 的反压机制在系统刚开始运行时便可以迅速提高系统的利用率。Redis 的加入，使得 Flink-N 一开始便拥有更高的 CPU 利用率，相较于 Flink 本身的反压机制，其最高优化率达到 36%，平均优化率为 21%。

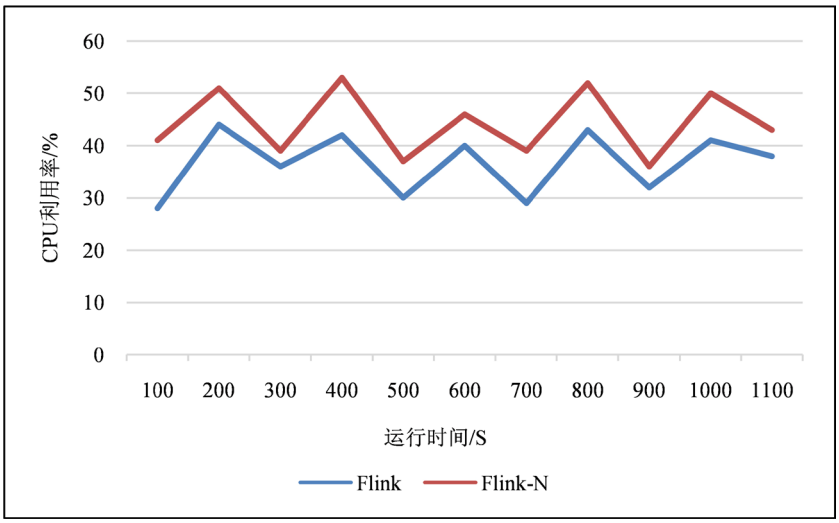

Figure 6. CPU utilization comparison
图 6. CPU 利用率对比

3) 传输时延效果评测

如图 7 所示为 Flink 与 Flink-N 的传输时延效果对比，随着时间增加，系统的吞吐量不断增大，伴随着系统吞吐量的增加，系统的传输时延不断增大。虽然在系统运行初始阶段，由于数据传输量小，但系统整体组件较多，Flink-N 也会出现时延较大的情况，但从整体角度来看，Flink-N 在时延方面的优化还是很成功的，其整体优化率达 18%，最大优化率达 23%。

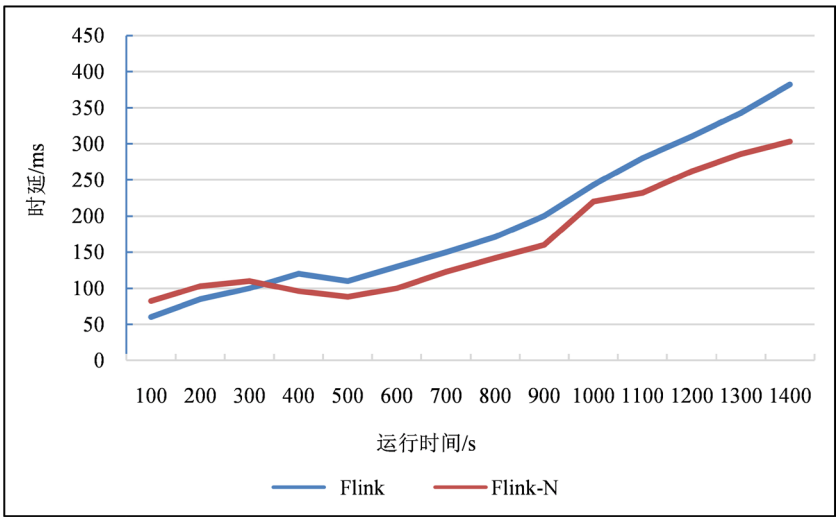

Figure 7. Time delay test comparison
图 7. 时延测试对比

7. 结论与展望

本文基于 Netty 水位机制，对 Flink 反压机制进行了动态水位调优，使 Flink 在面临远程传输问题时，具有更好的反压能力。经实验评测可知：优化后的 Flink-N 算法在吞吐量、CPU 利用率及时延方面均优于 Flink 默认的反压机制。但本文研究工作也有不足，可以从以下两方面改进：1) 对 Flink 本地传输进行

反压优化问题；2) 本文在 Flink-N 算法中接入了 Redis，是一种曲线式解决问题的方法。

基金项目

科技部重点研发项目“云计算和大数据”重点专项项目(2018YFB1004402)。

参考文献

[1] 袁海飞. 基于分布式实时计算架构的生产设备数据分析平台[J]. 电子技术与软件工程, 2020, 174(4): 217-219.

[2] 樊春美, 朱建生, 单杏花, 等. 基于 Flink 实时计算的自动化流控制算法[J]. 计算机技术与发展, 2020, 30(8): 66-72.

[3] https://tech.meituan.com/2018/10/18/meishi-data-flink.html

[4] https://help.aliyun.com/document_detail/110778.html?spm=a2c4g.11174283.2.6.114773d5r0epNT

[5] 姜红玉, 汪朋, 封雷. 基于流式计算的实时用户画像系统研究[J]. 计算机技术与发展, 2020, 30(7): 186-193.

[6] Aniello, L., Baldoni, R. and Querzoni, L. (2013) Adaptive Online Scheduling in Storm. *Proceedings of the* 7*th ACM International Conference on Distributed Event-Based Systems*, June 2013, 207-218. https://doi.org/10.1145/2488222.2488267

[7] Yang, M. and Ma, R.T.B. (2015) Smooth Task Migration in Apache Storm. *Proceedings of the* 2015 *ACM SIGMOD International Conference on Management of Data*, May 2015, 2067-2068. https://doi.org/10.1145/2723372.2764941

[8] 林琳. 阿里巴巴的大数据之路 JStorm 与 Blink 的发展史[J]. 计算机与网络, 2019, 45(2): 41-42.

[9] https://developer.aliyun.com/article/709397?spm=a2c6h.14164896.0.0.b095505ax2TAav

[10] 宋灵城. Flink 和 Spark Streaming 流式计算模型比较分析[J]. 通信技术, 2020, 53(1): 59-62.

[11] Mhand, M.A., Boulmakoul, A. and Badir, H. (2019) Scalable and Distributed Architecture Based on Apache Spark Streaming and PROM6 for Processing RoRo Terminals Logs. *Proceedings of the New Challenges in Data Sciences*: *Acts of the Second Conference of the Moroccan Classification Society*, March 2019, Article No. 19. https://doi.org/10.1145/3314074.3314093

[12] 关沫, 魏碧晴. 基于 Flink 框架的 TopN 堆排序优化算法[J]. 信息技术与网络安全, 2020(2): 23-26.

[13] 汪丽娟, 钱育蓉, 张猛, 等. 基于Flink平台的资源感知任务调度策略[J]. 东北师大学报:自然科学版, 2020, 52(2): 66-72.

[14] 何贞贞, 于炯, 李梓杨, 等. 基于 Flink 的任务调度策略[J]. 计算机工程与设计, 2020, 41(5): 1280-1287.

[15] Chen, C., Li, K., Ouyang, A., *et al.* (2016) GFlink: An In-Memory Computing Architecture on Heterogeneous CPU-GPU Clusters for Big Data. 45*th International Conference on Parallel Processing* (*ICPP*), Philadelphia, 16-19 August 2016, 542-551. https://doi.org/10.1109/ICPP.2016.69

[16] Chen, C., Li, K., Ouyang, A. and Li, K. (2018) FlinkCL: An OpenCL-Based In-Memory Computing Architecture on Heterogeneous CPU-GPU Clusters for Big Data. *IEEE Transactions on Computers*, **67**, 1765-1779. https://doi.org/10.1109/TC.2018.2839719

[17] Chen, C., Li, K., Ouyang, A., *et al.* (2017) GPU-Accelerated Parallel Hierarchical Extreme Learning Machine on Flink for Big Data. *IEEE Transactions on Systems*, *Man*, *and Cybernetics*: *Systems*, **47**, 2740-2753. https://doi.org/10.1109/TSMC.2017.2690673