软件学报 ISSN 1000-9825, CODEN RUXUEW E-mail: jos@iscas.ac.cn
Journal of Software, [doi: 10.13328/j.cnki.jos.000000] http://www.jos.org.cn

面向 Apache Flink 流式分析应用的高吞吐优化技术*

秦政[1,2], 许利杰[1,2,3,4], 陈伟[1,2,3,4], 王毅[1,2],吴铭钞[1,2], 曾鸿斌[1,2],王伟[1,2,3,4]

[1](中国科学院软件研究所,北京 100190)

[2](中国科学院大学,北京 100049)

[3](计算机科学国家重点实验室,北京 100049)

[4](中国科学院大学南京学院,南京 211135)

通讯作者: 许利杰, E-mail: xulijie09@otcaix.iscas.ac.cn

摘 要: 随着大数据时代的到来,海量的用户数据赋能了众多数据驱动的行业应用,例如智慧交通、智能电网、商品推荐等.在数据实时性要求高的应用场景下,数据中的业务价值会随时间快速降低,因此数据分析系统需要具有高吞吐和低延迟能力,以 Apache Flink 为代表的流式大数据处理系统得到广泛应用.Flink 通过在集群的计算节点上并行化计算任务,水平扩展系统吞吐量.然而,已有研究指出,Flink 存在单点性能弱,集群水平可扩展性差的问题.为了提高流式大数据处理系统的吞吐量,研究者在控制平面设计、系统算子实现和任务间信息共享等方面开展优化,但尚缺乏对流式分析应用数据流的关注.流式分析应用是由事件流驱动并使用有状态处理函数的应用,例如智能电网场景下的低电压检测应用、商品推荐场景下的广告活动分析应用等.本文对典型的流式分析应用的数据流特征进行分析,总结其中存在的三个水平可扩展性瓶颈并给出相应的优化策略,包括: 键级水位线策略,动态负载分发策略和低开销跨节点数据交换策略.基于上述优化技术,本文对 Flink 框架进行扩展形成原型系统 Trilink,并应用于低电压检测应用,桥梁拱顶监测应用和 Yahoo Streaming Benchmark.实验结果表明,相较于原生 Flink, Trilink 在单机环境下吞吐率提升 6 倍以上,8 节点下水平扩展加速比提高 1.6 倍以上.

关键词: 流式处理,分布式系统,性能优化,大数据系统

中图法分类号: TP311

中文引用格式:秦政,许利杰,陈伟,王毅,吴铭钞,曾鸿斌,王伟. 面向 Apache Flink 流式分析应用的高吞吐优化技术.软件学报,2024

英文引用格式: Qin Z, Xu LJ, Chen W, Wang Y, Wu MC, Zeng HB, Wang W. High Throughput Optimization Techniques for Apache Flink. Ruan Jian Xue Bao/Journal of Software, 2024 (in Chinese).

High Throughput Optimization Techniques for Apache Flink

QIN Zheng[1,2], XU Li-Jie[1,2,3,4], CHEN Wei[1,2,3,4], WANG Yi[1,2], WU Ming-Chao[1,2], ZENG Hong-Bin[1,2], WANG Wei[1,2,3,4]

[1](Chinese Academy of Sciences, Institute of Software Chinese Academy of Sciences, Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

[3](State Key Laboratory of Computer Science, Beijing 100049, China)

[4](University of Chinese Academy of Sciences,Nanjing 211135, China)

Abstract: With the advent of the big data era, massive volumes of user data have empowered numerous data-driven industry applications, such as smart grids, intelligent transportation, and product recommendations. In scenarios where real-time data is crucial, the business value embedded within data rapidly diminishes over time. Consequently, data analysis systems require high throughput and low latency.

* 国家重点研发计划(2021YFB2600301).
National key Research and Development Plan (2021YFB2600301).

Stream-based big data processing systems, exemplified by Apache Flink, have been widely adopted. Flink enhances system throughput by parallelizing computing tasks across cluster nodes. However, existing research indicates that Flink suffers from single-point performance weaknesses and poor cluster scalability. To improve the throughput of streaming big data processing systems, researchers have focused on optimizations in control plane design, system operator implementation, and inter-task information sharing. However, there is still a lack of attention to the data flow in streaming analysis applications. These applications, driven by event streams and employing stateful processing functions, include low voltage detection in smart grids and advertisement campaign analysis in product recommendations. This paper analyzes the data flow characteristics of typical streaming analysis applications, identifying three scalability bottlenecks and proposing corresponding optimization strategies: Key-level Watermark Strategy, Dynamic Load Distribution Strategy and Low-Overhead Data Exchange Strategy. Based on these optimization techniques, this paper implements Trilink based on Flink and applies it to low voltage detection applications, bridge arch crowns monitoring application and the Yahoo Streaming Benchmark. Experimental results show that compared to native Flink, the modified system, Trilink, achieves more than a 6-fold increase in throughput in a single-machine environment and over a 1.6-fold improvement in horizontal scaling acceleration in an 8-node setup.

Key words: Streaming processing, Distributed system, Performance optimization, Big data system

1 引言

随着互联网的快速发展,特别是物联网、移动计算等技术的广泛应用,人类社会进入“万物互联”时代,由此产生的数据规模快速膨胀.海量数据赋能了众多数据驱动的行业应用,包括智能交通、智慧城市、智能电网、智慧物流、商品推荐等,数据已经成为重要的生产要素[1][2].在上述广泛业务场景下,数据实时产生、传递并等待分析.

流式数据[3]指持续产生、没有边界的数据,如物联网设备上的传感器数据和网页上的广告点击事件等.流式数据准确地刻画现实世界连续不断的动态变化,蕴含着丰富的分析价值.流式数据通常由流式处理系统进行实时分析,通过持续的计算任务快速地对数据进行处理并给出响应.例如,在交通基础设施数字化中的隧道施工场景下,可以通过传感器实时采集拱顶沉降数据并进行处理分析,提高监测和预警能力,从而增强施工安全保障.另一方面,流式数据往往具有流速大,流量快的特点.例如,在典型电网场景中,数据以每秒十万级的速度流入中央处理节点;FaceBook 每小时的图片上传量约 15000000 张[4];阿里巴巴在线交易每秒产生约 580000 条实时交易数据[5],对系统性能提出了极高的要求.

流式处理系统 Apache Flink[6]已经被广泛采用,并在多个领域证明了其价值[7][8][9][10].Flink 能够通过水平扩展,有效增加系统处理能力,满足日益增长的数据处理需求.然而,尽管现有的流式处理技术在性能和可用性方面取得了显著进步,但它们在可扩展性方面仍存在不足,特别是在处理高速数据流或面临高复杂度计算时,系统可能会面临性能瓶颈问题.例如,计算负载的不均衡会导致单点性能瓶颈和整体吞吐量下降.已有研究[11]指出,以 Yahoo Streaming Benchmark[12]为例的 Flink 流式数据分析应用的水平可扩展性随节点数增多加速比逐渐下降,从 1 节点拓展到 4 节点吞吐率提升 1 倍,但是从 4 节点扩展到 16 节点时,吞吐率没有任何提升.

海量数据及其规模的持续增加对流式数据处理系统的性能提出了更高的要求.因此,针对流式处理系统的水平可扩展性进行优化具有实际的研究意义,这不仅可以进一步提高系统处理能力,也可以在同等计算能力下有效节约硬件资源.已有优化工作尝试从动态资源分配和任务调度等方法提高 Flink 水平扩展性从而提高系统吞吐,然而它们往往需要针对特定的计算任务和流式数据分布特性进行配置,在面对动态变化的实时数据流时,其存在着较明显的滞后性和较大的计算开销.

本工作在分析典型流式分析应用的基础上进行通用数据流建模,从数据流视角观察分析流式处理过程中可能存在的瓶颈问题,并针对性的通过测试和分析进行验证;进一步的,结合测试分析结果,设计一系列的优化机制和策略应对水平扩展的瓶颈问题,提高流式处理系统性能.

首先,通过对多个典型流式处理应用的数据流进行分析,论文发现显著影响 Flink 流式处理系统的实时处理和水平扩展能力的问题包括:

(1) 数据乱序耦合:在流式数据的产生、采集和传输中,存在不可避免的数据乱序,造成计算结果的不确定性.为保证计算正确性,现有 Flink 系统采用子任务级水位线机制,当数据达到水位线阈值后再进行计算消费.然

而,在大规模实时数据场景下,多个分区被并行消费时,来自这些分区的事件会发生耦合,造成数据间额外的乱序,带来额外的等待时间或更严重的时序耦合,导致系统的吞吐率下降.当节点数增加时,乱序数据的耦合也会增加,造成 Flink 的可扩展性下降.

(2) 节点负载不均:Flink 需要将计算负载进行划分以在多个计算节点并行计算.当前,Flink 使用受到广泛验证的 murmurHash 作为分配依据,将数据分配到键值组和对应的算子子任务上.然而现有工作和实验分析表明,负载的划分和分发存在严重不均衡,造成系统吞吐量的下降和轻负载节点上的资源浪费.

(3) 跨节点数据交换:由于计算任务的算子间的数据传输使用内存传输和网络通信传输混合的模式,且构成计算任务的算子可能分布于多个节点,随着数据量增加,节点数量增加,这一问题会愈发严重,因而 Flink 可扩展性受限,吞吐率下降.

针对以上三个导致性能瓶颈的根因,本文设计和实现对应的优化策略:

(1) 键级水位线策略:当不同数据源产生的数据具有不同的逻辑主键时,由于网络时延地理位置等不确定因素，子任务级的水位线机制会导致不必要的时序耦合.因此,针对原有粗粒度的水位线机制,本文设计键级水位线机制和相应的系统算子,从而检测并减少乱序数据比例,提高系统吞吐量.

(2) 动态负载分发策略:本文扩展 KeyBy 算子的负载分发策略,设计了分别适用于四种不同数据分布场景的数据分发新策略;基于负载均衡状态在线监测提出运行时负载分发策略的动态调整机制,从而实现流式处理系统的自适应动态负载均衡.

(3) 基于键值的跨节点数据交换优化策略:为了尽可能减少或消除跨节点的数据交换,本文设计了三种基于键值的数据交换最小化策略, 减少或避免节点间的全量数据传输,以提高吞吐量并降低传输延迟, 并实现对应的系统算子.

本文基于 Flink 进行了上述优化策略的设计实现,形成了原型系统 Trilink,并通过实验验证了方法的有效性.总体来说,论文工作的贡献如下:

(1) 基于典型流式分析应用的分析构建了通用的流式处理数据流模型,通过分析测试验证了导致分布式流式处理系统吞吐率瓶颈的三个潜在的水平扩展性根因问题;

(2) 从数据流处理的角度出发,针对发现的三个水平扩展性问题设计了一系列优化策略,包括: 键级水位线、面向多种数据分布场景的动态负载分发和基于键值的跨节点数据交换优化,实现时序化保证、负载均衡和数据传输多维度的优化,有效提高流式处理系统在分布式环境下的水平可扩展性;

(3) 基于 Flink 实现上述优化策略,设计实现面向高吞吐的流式处理系统原型 Trilink,并在测试基准和真实应用数据集上开展实验评估.实验结果表明,相较于原有 Flink, Trilink 单机吞吐量提升 6 倍以上,八节点集群下水平扩展加速比提升 1.6 倍以上.

本文第 1 节介绍流式处理系统及其优化技术相关工作.第 2 节介绍相关研究工作.第 3 节介绍基于典型流式分析应用的数据流特征与性能瓶颈分析.第 4 节提出并设计高性能流式处理的一系列优化策略,并在第 5 节展示了相应的系统实现.第 6 节基于测试基准和真实应用对论文提出的方法进行实验评估,以验证本文方法的有效性.第 7 节总结全文.

2 相关工作

2.1 流式处理系统

流式查询的概念最早源于 Tapestry[13]系统,流式处理系统历经三个阶段的发展[14].第一代流式处理系统,也被称为流式数据管理系统,由数据库系统和规则引擎组成.用户通过定义规则中的触发条件和触发动作,当新数据进入系统时,如果满足规则的触发条件,触发动作会被执行以修改系统的内部状态.典型的第一代流式处理系统包括 TelegraphCQ[15]、Spade Stream Processing Engine[16].

第二代流式处理系统已经开始专门用于处理流式数据,也具有更直观的流式处理语义.斯坦福大学研发的 STREAM[17]在 SQL 上实现具有流式语义的 CQL (Continuous Query Language);Aurora[18]使用有向无环图来表

示流式计算过程,图上的节点定义操作命令,节点间的边表示数据流向和操作顺序;.STREAM 和 Aurora 都是单机系统,Borealis[19]对 Aurora 在分布式集群上扩展实现,使其能利用集群上的多个计算节点并行计算.

MapReduce[20]编程模型的出现深刻影响了大数据与分布式计算领域,出现了第三代流式处理系统,即被广泛使用的分布式流式处理系统,包括 Storm[21]、Spark Streaming[22]、Flink.这些流式处理系统都利用分布式集群上的计算节点并行计算,使用有向无环图描述计算过程,具有着相似的计算模式和流程,其典型特征如表 1 典型流式系统所示.其中,Flink 是一个流批一体、高效、分布式的处理引擎,支持连续数据的流式处理. Flink 支持状态管理和灵活的窗口机制,支持“恰好一次”交付保证,通过异步快照机制支持错误容忍[23].

目前,Flink 已经连续两年蝉联 Apache 社区最活跃项目,并被绝大多数的互联网企业作为流计算的事实标准来采用,包括美团、字节跳动等.因此,本文将 Flink 作为主要的研究对象,并将相关的优化设计基于 Flink 完成原型系统实现与实验.需要注意的是,由于第三代流式处理系统具有相似的数据处理模式和流程,因此本文设计的优化策略具有一定的通用性.

表 1 典型流式系统

	Storm	Spark Streaming	Flink
处理模式	连续流	微批	连续流/微批
交付保证	至少一次	恰好一次	恰好一次
状态管理	无	有	有
吞吐量	低	高	高
延迟	毫秒级	秒级	毫秒级

2.2 流式处理系统性能优化

面对流式处理应用吞吐量低和处理延迟高的问题,已有的研究从垂直可扩展性、控制平面、系统算子实现、多查询信息共享、水平可扩展性五个方面对系统吞吐量和处理延迟进行优化.与已有工作不同,本文从数据流的角度出发,不关注于局部特征优化,从整体计算流程的角度尝试分析现有系统瓶颈并提出优化策略,优化系统整体吞吐能力,同时降低处理延迟.

2.2.1 面向垂直可扩展性的研究

Zeuch 等人[24]对流计算框架在单机上的性能表现和性能瓶颈展开研究,发现 Flink 等现代流式处理系统并不能充分发挥硬件的性能,Flink 仅能发挥硬件理论性能的 1.5%,而使用更接近底层硬件的 C++实现相同的计算任务,可以更充分利用机器性能,使用无锁队列等设计对实现进行优化,可以更进一步逼近理论极限,达到理论性能的 91.4%,相当于 Flink 的单机性能的 61.4 倍.但该研究聚焦于单机上的性能瓶颈和垂直可扩展性,没有关注于分布式情况下的性能瓶颈和水平可扩展性,而实际应用中分布式的部署和运行模式已成为当前流计算系统的主流.

Saber[25]是一个混合架构的流式处理系统,支持在 GPU 上执行流计算任务,允许在 CPU 和 GPU 上混合调度执行流计算任务.实验表明拥有大量流处理器的 GPU 的吞吐量往往高于 CPU,而同时调度使用 CPU 和 GPU,在多种任务下能比单独使用 CPU 或 GPU 具有更高的吞吐量.但 Saber 关注于基于窗口的流式 SQL 处理,不能覆盖其他的流计算任务,也不适用于无 GPU 的计算集群.

2.2.2 面向流式处理系统控制平台的研究

Chi[26]提供一个控制平台,对 Flink 流计算系统进行持续的监控,允许在线动态重配置系统参数,例如算子并行度.Chi 引入一种新的响应式编程模型和设计机制,采用异步执行控制策略,避免全局同步.相关实验表明,在动态扩缩容和错误恢复时,Chi 相对于 Flink 具有更高的吞吐量,且能更快地恢复低处理延迟.但 Chi 没有考虑数据通路上的问题,例如数据传输所导致的网络瓶颈.同时,Chi 需要用户学习其编程模型并对目标场景进行实现和适配,使用门槛较高.

Varga[27]和 Arkian[28]关注于 Flink 集群计算资源的自动扩缩容,相对稀缺的计算资源和不平稳的工作负载使得不可能为每个应用程序分配一组静态资源.自动扩缩容可以保证流式处理应用保持足够的吞吐量来处理高峰时的工作负载,同时不会使用超过严格限制的资源,也可以在工作负载较低时释放不必要的计算资源.自动扩缩容关注于集群视角下资源的调度,而非有限计算资源下的性能优化,而且自动扩缩容的过程伴随流式计算任务的重启,会导致扩缩容过程中的系统不可用.

He 等人[29]对异构集群上的 Flink 数据划分问题进行研究,关注异构集群下不同机器的硬件资源差异和其他评价指标,通过指标进行评分以确定在数据划分网络图中边的权值,从而调整异构集群上负载分布.但该工作没有关注 Flink KeyBy 算子上因基于哈希分配策略导致的负载均衡问题,且没有关心算子节点上的状态,需要最终在单一节点上聚合结果,可能存在单点故障和单点性能瓶颈问题.

2.2.3 面向系统算子实现的研究

Flink 在处理有顺序要求的滑动窗口聚合算子时,如果存在数据乱序,需要把数据聚合到如增强红黑树等数据结构中,以便后续查询和使用.Tangwongsan 等人[30]提出一种新的通用乱序滑动窗口聚合数据结构 FiBA.FiBA 相比 Flink 内部实现的滑动窗口聚合数据结构,拥有更低的时间复杂度和数量级上的吞吐量提升,也因此提升了滑动窗口上的聚合算子的性能.

流数据动态变化的特性使得传统的索引结构不再适用.Shahvarani 等人[31]通过对滑动窗口内的流数据建立索引来提升窗口连接的性能,他们提出树状结构的索引以及对应的并发控制的机制,使得该索引可以在多线程共享情况下高效更新.在此基础上,他们设计实现基于索引的并行流连接,从而可以更好地利用多核处理器的计算能力.

FiBA 和滑动窗口索引只能提升滑动窗口场景下的性能,无法解决其他场景下的性能问题,例如 Yahoo Streaming Benchmark 中使用有状态处理函数的流式分析场景.

2.2.4 面向多查询信息共享的研究

现代流式处理系统主要用来处理在流式数据上的长时间特定查询分析,这些特定的查询可以手动优化以提高性能.然而,即时查询场景不仅包括长时间的查询分析,也有数千个短时间的临时查询分析.AStream[32]针对即时查询场景进行优化,通过在多租户环境下共享资源加速短时间的临时查询分析,并采用共享的算子来避免冗余计算.AJoin[33]在 AStream 的基础上进一步优化,减小算子共享的代价,重优化运行时查询计划,支持运行时动态扩缩容.

现代流式处理系统的增量处理和视图维护在多个查询之间独立,这会给相同输入流上的并发增量查询带来冗余和不必要的开销,使得每个查询必须在相同的输入流上独立地维护相同的索引状态,而新的查询必须从头开始构建这种状态,然后才能产生第一个结果.McSherr 等人[34]引入共享的视图及索引,从而在不影响数据并行和可伸缩性的情况下,允许并发查询重用相同的内存状态.多查询优化适用于并发产生的 SQL 查询,通过共享多个查询间的信息减少冗余计算.然而,现代流式处理系统主要处理的长时间特定查询分析无法因此而受益.

2.2.5 面向水平可扩展性的研究

流式处理系统能够通过增加更多的处理节点来提高处理能力.在基于 Flink 的水平可扩展性优化技术中,研究主要集中于动态资源分配和任务调度等关键领域.

Zhang[35]等人提出了一种新的动态滑动时间机制来检测延迟到达的数据流.该窗口机制主要包括:自适应性窗口标记位,迟到数据检测和窗口内时序校正.实验显示,该方法在乱序数据流下能够有效检测并降低系统处理时延.

Yue[36]等人提出了一种基于运行时间预测的动态资源分配策略 RABORP,制定实施动态资源分配计划.通过预测各个迭代超步的运行时间,在迭代作业提交时和超步同步屏障处分别进行资源的初始分配和动态调整,以保证可使用最小资源集合,并保证迭代作业在用户规定的运行时限内完成.实验结果表明,其有效提升了 Flink 系统性能.

Salman[37]提出了基于网格的索引技术,从而实现连续流式数据的高效处理,并基于 Flink 实现了原型系统

GeoFlink.GeoFlink 基于网格索引定义流式数据的相似性,从而实现均匀数据分区.GeoFlink 为相近数据分配相同的键,并基于 Flink 的 KeyBy 算子实现数据分发.Salman 认为相似数据具有相近处理逻辑,涉及相似的算子,从而降低计算图的复杂度.实验表明,GeoFlink 相较于 Flink 表现出了显著的吞吐率与可扩展性提高.

3 典型流式分析应用的数据流特征与性能瓶颈分析

3.1 典型流式分析应用数据流特征

流式分析处理在物联网、交通等多种场景中得到广泛应用.本文选取流式分析应用进行数据流特征分析,典型应用的选取标准包括: (1) 真实业务场景下的流式处理应用; (2) 具有普遍认可的流式数据处理行业基准; (3) 在社会重要领域具有重大价值.因此,论文选择智能电网场景下的低电压检测、广告活动分析场景下的 Yahoo Streaming Benchmark 和智能出租车调度场景下的疲劳驾驶提醒作为典型的流式分析应用,其中:

(1) 智能电网场景下的低电压检测应用涉及海量物联网传感器设备,要求系统能够处理海量、高速生成的数据,而且对实时性和准确性有极高的要求,这一场景能够展示流式处理系统在处理大规模物联网数据方面的能力;

(2) 作为业界公认的流式处理分析测试基准,Yahoo Streaming Benchmark 提供了一个被广泛接受和验证的性能评估标准;

(3) 智能出租车调度中的疲劳驾驶提醒[38]关注于实时的交通数据处理和应急响应机制,体现流式系统处理复杂事件和实时决策的能力,是一个对数据分析实时性要求极高的应用,充分体现了流式处理在社会重要领域的应用价值.

下面本工作将对上述三个应用分别进行介绍并构建数据流,并总结出典型流式分析应用的通用数据流.

- **低电压检测**

低电压检测应用根据数以百万计的电压传感器的历史数据和实时数据,检测电网中的电压状态,从而实现智能调控和告警.如图 1 低电压检测数据流图 所示,数据从分散各地的电压传感器传输到分布式消息队列 Kafka[39],存储在 Kafka 的不同分区内.每个分区被视为数据源的一个分片,由连接器的读取单元读入.由于低电压检测需要分析每个传感器的电压状态,通过 KeyBy 算子进行数据交换,聚合具有同一传感器 ID 的数据到同一处理函数,处理函数根据规则更新状态,输出结果.低电压检测应用使用传感器数据产生时的事件时间作为时间语义,使用水位线和定时事件统计特定时间段的状态结果.

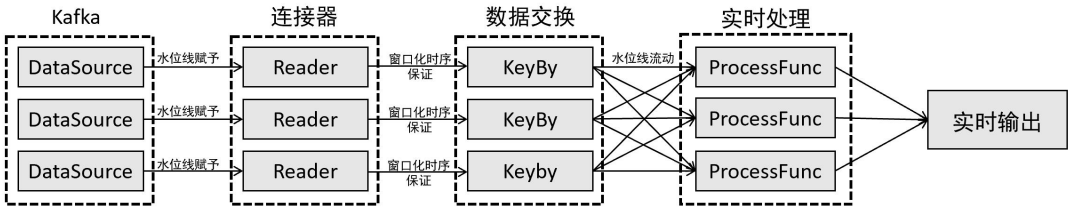

图 1 低电压检测数据流图

- **Yahoo Streaming Benchmark**

Yahoo Streaming Benchmark 根据广告活动事件流统计广告宣传活动的观看数量,以实时反馈广告活动的效果.如图 2 所示,数据存储在 Kafka 的不同分区内,经连接器的读取单元读入后,反序列化为广告事件对象,过滤出广告观看事件,投影得到目标属性,与 Redis[40]中的数据进行连接获得活动 ID,由于 Yahoo Streaming Benchmark 需要统计同一活动下的一定时间范围内的广告观看事件,需要通过 KeyBy 算子进行数据交换,聚合具有相同活动 ID 的数据到同一处理函数,处理函数根据规则更新状态,输出结果.Yahoo Streaming Benchmark

使用广告事件的实际发生时的事件时间作为时间语义,但不依赖于定时事件,而是自行存储和输出各个窗口内的状态.

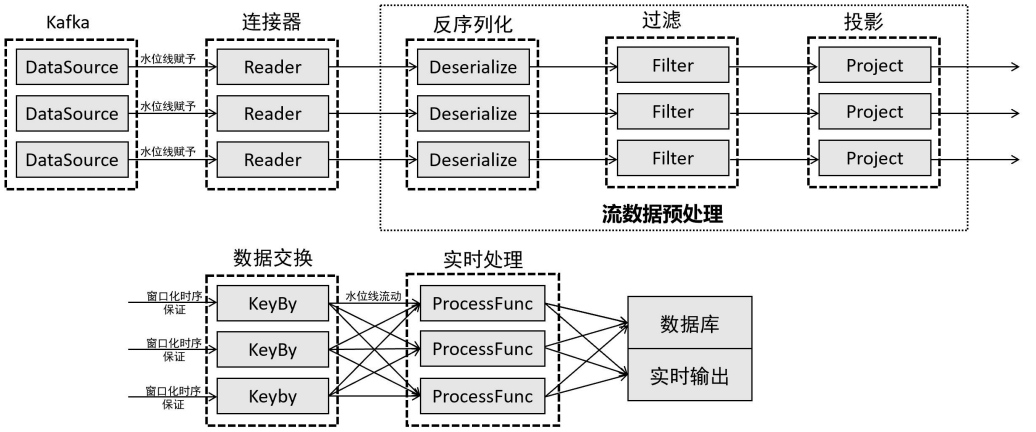

图 2 Yahoo Streaming Benchmarking 数据流图

- **疲劳驾驶提醒**

疲劳驾驶提醒应用根据出租车运营事件流检测过长时间运营的出租车,以提醒司机和智能调度出租车.数据存储在 Kafka 的不同分区内,经连接器的读取单元读入后,过滤出行驶中车辆,由于疲劳驾驶提醒需要根据同一出租车运营事件 ID 统计运营时间信息,需要通过 KeyBy 算子进行数据交换,聚合具有相同事件 ID 的数据到同一处理函数,处理函数根据规则更新状态,并输出结果.疲劳驾驶提醒应用使用运营事件实际发生时的事件时间作为时间语义,使用水位线和定时事件在过长时间疲劳驾驶后触发疲劳驾驶提醒.该应用的数据流图如下图 3 疲劳驾驶提醒数据流图所示.

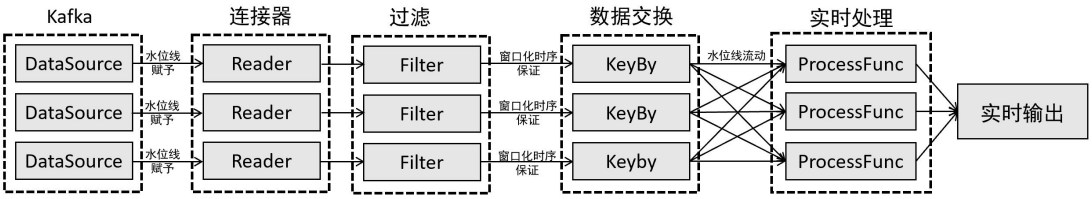

图 3 疲劳驾驶提醒数据流图

观察分析可发现，虽然应用场景和领域不同，但是上述三个典型流式分析应用都在消息源处赋予数据水位线并将水位线顺序流动,使用数据源连接器根据消息队列的不同分区并行读入数据,通过 KeyBy 算子聚合同键数据,使用有状态处理函数进行统计分析,这三个应用的对比分析如表 2 典型流式分析应用对比分析所示.

基于上述典型流式分析应用的分析比较,本工作对数据流图进行总结抽象, 如图 4 典型流式分析应用数据流图所示.在此架构中,数据源生成的实时数据首先通过 Kafka 流入流式处理系统,系统中引入水位线机制以保障数据的时序化处理;随后,通过 KeyBy 算子,数据被分发至各个处理节点,以执行具体的实时处理和计算任务,并最终输出结果.算子之间顺序链接,形成流水线式实时处理架构.

另一方面,流式处理系统往往通过并行化计算任务水平扩展系统性能,但是水平扩展瓶颈问题确实存在,并导致系统吞吐量不能随着硬件资源的增加而有效提升.如图 5 所示,Yahoo Streaming Benchmark 从 1 节点拓展到 4 节点吞吐量仅提升 1 倍,从 4 节点扩展到 8 节点时,吞吐量甚至没有任何提升;对低电压检测应用的水平

可扩展性的测试结果也类似,当增加到四倍硬件资源时,系统吞吐量仅增加 24.2%.

表 2 典型流式分析应用对比分析

	低电压检测	Yahoo Streaming Benchmark	Long Ride Alert
数据源	Kafka	Kafka	Kafka 或数据生成器
涉及算子	KeyBy	FlatMap、Filter、Project、KeyBy	KeyBy、Filter
使用 KeyBy	根据传感器 ID	根据广告 ID	根据运营 ID
处理函数	使用有状态处理函数	使用有状态处理函数	使用有状态处理函数
时间语义	事件时间	事件时间	事件时间
依赖水位线	是	是	是

对典型流式处理分析应用数据流进行观察分析,结合集群扩展加速比随节点数增加而逐渐下降这一现象,本工作总结分析了影响系统性能的潜在瓶颈.首先,系统通过计算和传递水位线来提高局部计算的相对正确性,水位线机制贯彻处理生命周期.然而,水位线机制会带来额外的等待时间或时序耦合,影响系统性能.此外,流式处理应用的核心功能主要围绕 KeyBy 算子展开,其涉及数据分发与节点间的数据传输,数据分发的均衡度以及节点间传输的效率,对系统的整体性能产生重大影响.

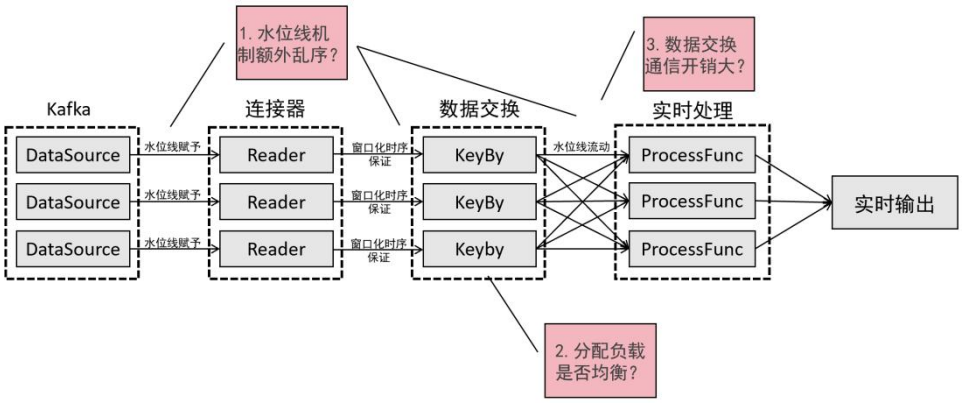

图 4 典型流式分析应用数据流图与潜在性能瓶颈

具体来说,经过观察分析,如图 4 所示,流式分析应用数据流中的潜在性能瓶颈包括:

(1) 在流式数据的产生、采集和传输中,存在不可避免的数据乱序,为保证计算正确性,流式处理系统往往采取子任务级水位线机制.水位线机制带来额外的等待时间或更严重的时序耦合,会造成系统的吞吐率下降.当节点数增加时,潜在的数据乱序耦合也会增加,会造成流式处理系统的可扩展性下降;

(2) 典型流式分析数据流中,通过 KeyBy 算子进行实时数据的分发,在分布式环境下,KeyBy 算子是否能够有效地将相当的计算负载分发至各个节点,会深刻影响到流式处理系统的性能;

(3) KeyBy 算子在进行数据分发时,涉及到大量的节点间通信和跨节点数据交换,带来造成额外的通信与传输开销,随着数据量增加,节点数量增加,这一问题会愈发严重,造成系统的吞吐率下降.

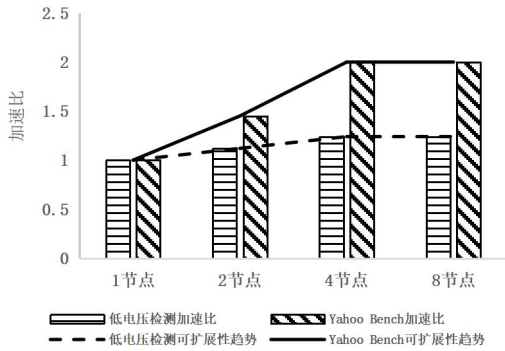

图 5 Flink 流式分析应用可扩展性测试图

论文工作分别对这三个问题进行相应的测试分析,以进一步验证推测结论的正确性.测试实验环境为 4 台服务器组成的集群,每台机器均具有一个型号为 Intel(R) Xeon(R) Gold 5215 @ 2.50GHz 的 CPU 和 128GB 内存,机器间通过千兆网络连接.4 台机器均部署 Flink,其中一台作为集群 Master.

为了测试典型流式分析应用的潜在瓶颈,本工作生成了一百万条数值数据进行简单的流式计算任务,其中,消息的主键为一组自增数字 ID,定义流式计算任务为对窗口大小为 5 的数据进行累加并输出结果.由于本文实验环境中 CPU 核数为 60,因而将 Flink 并行度默认设置为 60.

3.2 数据乱序耦合

流式处理系统往往采取一定的水位线机制以提高计算准确性.本工作首先介绍 Flink 的水位线机制,再通过一个简单的工业传感器案例展示 Flink 水位线机制造成的额外数据耦合;之后,通过对现有 Flink 系统进行测试,观察不同水位线大小造成的数据乱序程度及吞吐率下降程度.

Flink 采用水位线机制表示最大可容忍的事件延迟[41],水位线是每个算子子任务上的逻辑时钟,用于触发定时事件的计算.水位线在靠近数据源侧产生,在各个算子上依次向后传递,对于多输入算子,例如 KeyBy,会选择所有通道中的最小水位线.Flink 上的水位线是子任务级的,每个算子的子任务上都有唯一的水位线.当使用 Apache Kafka 作为数据源时,每个 Kafka 分区可能有一个简单的事件时间模式,例如时间戳单调递增.然而,多个分区被并行消费时,来自这些分区的事件发生耦合,破坏了分区内的相对有序.

例如,当输入数据来源于工业传感器数据时,各个传感器产生的数据在消息队列内往往是各自局部有序的[42],然而由于传感器部署位置导致的网络延迟差异,不同传感器产生的事件到达消息队列时不能保证事件时间具有严格递增关系[43].例如,传感器 A 在 T=3s 产生的事件到达消息队列的时间,可能晚于传感器 B 在 T=4s 产生的事件,也因此在队列中更位于尾部,造成了事实上的乱序耦合.另一方面,如果传感器存在本地时钟偏差,例如传感器 C 的本地时钟比传感器 D 早 10s,可能导致更严重的数据流乱序.

Flink 提供了子任务级有界水位线机制,允许正确处理乱序程度不超过最大乱序边界的流式数据,然而,当边界值设置较大时,即数据将额外等待较大时间时,可以有效降低数据乱序程度,但会造成系统输出时延提高,系统整体吞吐率下降.另一方面,当边界值设置较低时,即数据等待较少时间,那么潜在的数据乱序现象将会提高,为保证结果正确性,结果需要进行重计算并更新,系统吞吐率同样会下降.

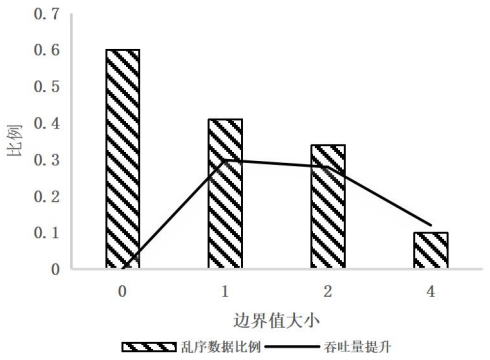

图 6 水位线边界设置乱序性能影响

本文通过实验分析了子任务级水位线边界设置对系统吞吐率和乱序程度造成的影响.实验设置三个数据源,其中一个数据源本地时钟存在 4s 的偏差,实验分别测试了边界值为 0s,1s,2s,4s 时的系统性能,结果如图 6 水位线边界设置乱序性能影响所示.实验结果表明,边界值的增加确实可以有效减少乱序数据比例,但这意味着更长的等待时间,带来了额外的时延.另一方面,当边界值过大时,即使乱序比例为 0,系统吞吐率也会受到显著下降,甚至造成堵塞.在现实场景中,本地时钟偏差和网络通信都具有随机性和不可知性,因此,系统性能会因为实时数据的质量产生巨大波动.在目前的典型流式分析应用中,大多数场景下不同数据来源间的直接聚合分析较少,全量数据来源间的聚合更少,当前的水位线机制粒度过大,造成了系统额外的时序耦合,带来了性能瓶颈.

3.3 KeyBy分配负载不均衡

分布式流式处理系统需要将任务负载分发到多个计算节点上并行处理.负载不均衡问题会造成系统吞吐量的下降和轻负载节点上的资源得不到充分利用.本小节通过实验测试论证了现有典型流式系统 Flink 在进行实时计算时,KeyBy 算子确实存在负载分配不均衡的现象.

Flink 上的数据流往往使用 KeyBy 进行数据交换,同时将任务并行化,分配负载到多个执行节点上并行处理.当前,KeyBy 使用受到广泛验证的 murmurHash[44]作为分配依据,将数据分配到键值组和对应的算子子任务上.

在基于 Flink 的 KeyBy 算子负载均衡情况测试实验中,消息的主键为自增的数字 ID,并行度为 60,总数据量为一百万条.理想情况下,60 个算子的子任务都应处理 $10^6 \div 60 \approx 16666$ 条数据,即 1.67%的负载.然而,实验结果如表 3 Flink KeyBy 算子负载均衡测试所示,13.3%的子任务承担 2.17%以上的负载,而超过一半的子任务仅承受不足 1.57%的负载,最高负载节点甚至比最低负载节点承受额外 111.0%的负载,这使得最高负载节点相比负载均衡情况下计算时间增加 60.8%,导致系统整体吞吐量受限和轻负载节点的资源浪费.

表 3 Flink KeyBy 算子负载均衡测试

负载百分比(%)	$1.27 \le x \le 1.57$	$1.57 < x \le 1.87$	$1.87 < x \le 2.17$	$2.17 < x \le 2.47$	$2.47 < x \le 2.77$
子任务数量	33	19	0	4	4
所占资源比例	55%	61.7%	0%	6.7%	6.7%

为了衡量负载均衡的状况,本文定义负载均衡度和超额计算时间两个指标及其计算公式如下:

$$\text{负载均衡度} = \frac{\text{最低负载节点处理数据量}}{\text{最高负载节点处理数据量}} \tag{1}$$

$$\text{额外计算时间} = \frac{\text{最高负载节点处理数据量} - \text{总数据量}/\text{节点数}}{\text{总数据量}/\text{节点数}} \tag{2}$$

其中,负载均衡度在[0, 1]之间,越接近 0,表明系统负载越不均衡;越接近 1,表明系统负载均衡状况越好.额

外计算时间反映因系统的负载不均衡导致的最高负载节点的额外计算时间,也反映系统整体吞吐量的受限情况.

如图 7 KeyBy 算子负载均衡测试所示,随着主键数量增大,系统负载趋向于均衡,但即使全量数据均有独立 Key 时,负载均衡度也仅有 0.47.节点间负载不均将造成计算资源的不充分利用,造成系统吞吐下降.另一方面,由于负载不均,子任务间计算进度将出现差异,若后续计算流程中包含跨任务间聚合等计算时,会造成额外的等待时延.

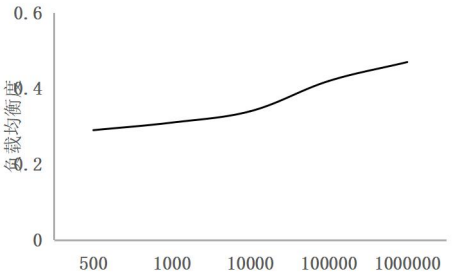

图 7 KeyBy 算子负载均衡测试

3.4 跨节点数据交换

流式分析结果往往与数据主键相关联,Flink 使用 KeyBy 进行基于键值的数据交换,从而将同一主键的数据聚合到同一计算节点上进行分析汇总.这种全量的数据交换可能使得网络带宽成为系统性能瓶颈,并增加数据的处理延迟,降低系统的吞吐率.

由于节点间的数据交换带来的性能开销很难直接度量,本文通过测试 Flink 系统在默认配置下,KeyBy 会造成的节点间交换数据量,以此来帮助判断 KeyBy 数据交换会给流式处理带来的额外性能开销.同样的,本实验中采取键值均匀分布的数据执行 3.2 节中的计算任务,通过识别经过 KeyBy 后数据是否仍在原有节点来进行数据交换量计算.

表 4 KeyBy 节点间数据交换量

数据量	10000	10000	10000	1000000	1000000	1000000
节点数	1	2	4	1	2	4
通信比例	0%	51%	75%	0%	48%	74%

如表 4 KeyBy 节点间数据交换量所示,实验发现当前 KeyBy 算子在进行基于键值的数据交换时,需要高比例的数据通信与传输,这将影响系统的处理性能;尤其当计算节点间网络带宽有限且数据流速快时,系统吞吐率和扩展性会产生严重下降.

4 高性能流式处理性能优化设计

本节针对第 3 节分析发现的影响流式处理系统性能的三个瓶颈问题分别提出了优化技术.

4.1 键级水位线设置

本工作经实验发现,现有子任务级水位线机制会造成流数据时序耦合,严重影响系统的吞吐率.在典型的流式分析应用,例如低电压检测应用中,传感器的数据间跨片区或跨城市聚合操作较少,但由于水位线是子任务级的,在存在网络延迟差异或本地时钟偏差的场景下,即使各传感器数据相对有序,整体的数据流在 Flink 中也被视为具有严重的数据乱序.在要求保证结果正确的情况下,需要基于迟到数据更新结果,这些额外的更新操作会

导致系统吞吐量的下降.

当前 Flink 中水位线的粒度是以子任务为单位的.这种设计在处理单一数据源时效果良好,但在面对多数据源时可能存在问题.这是因为,不同的数据源可能有不同的数据产生频率和时间特性.当所有这些数据源被同一个粗粒度的子任务处理时,可能会发生数据耦合,其中低频率数据源可能拖延整个数据流的处理速度.随着数据源增多以及计算节点增加,耦合程度进一步加深,显著影响到系统的可扩展性.粗粒度的水位线可能导致高频率数据源的数据被不必要地延迟,影响系统的实时性和吞吐率.例如,图 8 水位线时序图中,对于每一条流数据(k,v,t),k 代表键值,v 代表数据,t 为该数据流入系统时间戳,假设处理时间窗口为 2.图 8 中,当 Flink 接入 t = 3 数据(k_1,v,3)时,对所有 t 不大于 2 的数据进一步计算处理.当迟到数据(k_1,v,1)出现时,为保证计算结果正确性,需要对已计算的 3 条数据进行重计算.此外,由于数据处理不够及时,系统的缓冲区可能会被迅速填满,进而影响到系统的稳定性.实时流处理场景中数据量大且多样化,这一问题将变得尤为显著.因此,当前 Flink 中水位线机制粒度过大,不适用于大量典型流式处理场景.

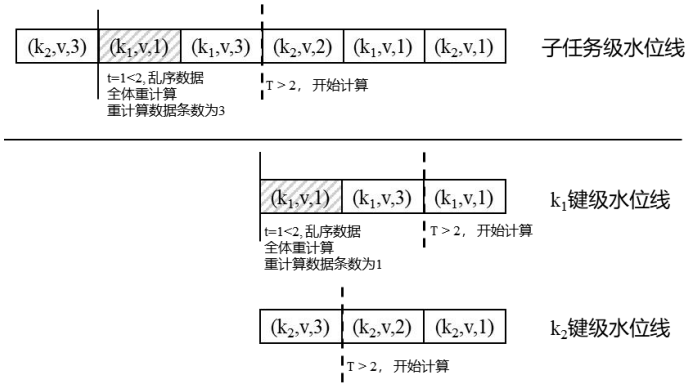

图 8 水位线时序图

本文基于分治思想提出键级水位线设计,即对每一主键分别赋予独立管理的水位线机制.具体来说,对于每一个不同的主键,系统均维护与之相匹配的水位线,可以最大程度降低因数据传输和并行化处理带来的额外时序耦合.本文提出的键级水位线如图 9 主键级水位线示意图所示,键级水位线与主键相关联.每个算子子任务上具有对应水位线信息,每个主键相关的水位线随着数据在算子间传递.对于不同数据产生源间数据乱序,键级水位线相比子任务级水位线而言:一方面,使用子任务级水位线可能需要长时间等待所有事件的到达,从而导致大量中间状态的保持,键级水位线可以减少这种不必要的状态保持,因为每个键的窗口可以独立于其他键关闭和清理;另一方面,由于每个键的水位线独立更新,窗口操作可以更快地完成,从而对于有实时性要求的应用,可以实现更低的处理延迟.键级水位线提供了一种更精细的控制方法,可以减少数据的乱序问题.即使不同数据源通信效率差异造成生成时间与到达时间间的紊乱,系统也会基于计算任务需求和相应的键级水位线进行合理等待,不会带来额外的乱序处理操作.图 8 中,当 Flink 接入 t = 3 数据(k_1,v,3)时,只需要对该主键所涉及到的 1 条数据进行重算,减小了计算和等待开销.键级的水位线在该场景下,可以减少迟到数据比例,从而减少基于迟到数据的更新操作,提高系统吞吐量.

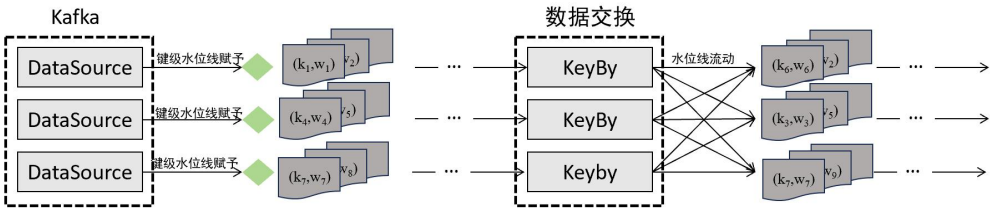

图 9 主键级水位线示意图

本文设计了有状态处理函数上的键级水位线机制,其核心是水位线的更新和定时事件的触发.不同于原有的有状态处理函数仅支持 Process 接口,用户需要在其中实现基于子任务级水位线的迟到元素检测和处理、非迟到元素处理,键级水位线基于设计模式中的模板模式,抽象分离出有状态处理函数中的基于主键级水位线的迟到元素检测、迟到元素处理、非迟到元素处理三部分逻辑.同时,也支持子任务级水位线上的迟到元素检测和处理,从而兼容原有的水位线语义,并可以根据子任务级水位线下的迟到元素数量与主键级水位线下的迟到元素数量,衡量性能的改善.为了检测并触发定时事件,本文基于红黑树存储定时事件,以实现对数时间复杂度的定时事件注册和常数时间复杂度的最早定时事件查询.

键级水位线机制适用于不同数据源间存在数据乱序、同一数据源内部数据相对有序、且各主键间计算任务相对独立的场景.该机制能够在保证结果正确性的前提下,避免不必要的乱序数据等待与重处理.

由于键级水位线会带来额外的空间和计算开销,该机制实际提供了一种最细粒度的控制方式,用户可以在键级和子任务级间自主调控.键级水位线机制同样支持 N-键级水位线,即 N 个 Key 共享同一水位线,从而达到空间开销与系统性能间的均衡.

4.2 KeyBy算子负载分发策略与动态负载均衡

KeyBy 算子在流处理中具有其特殊性,其根据键相关信息对流数据进行分区,对于分布式流处理框架来说,对其做优化适用于大部分的流处理应用.另一方面,KeyBy 算子涉及到状态的分区存储和管理,比如计算滑动窗口的平均值或者跟踪某个项的运行总数.因而,相较于传统的数据倾斜和负载均衡问题,KeyBy 算子的负载分发涉及到更多流式处理特性,包括实时窗口内数据分布信息和历史状态信息.本节首先针对常见的数据分布场景和集群配置情况提出了四种负载分发优化策略;接着,设计了在线监测机制与负载均衡策略动态配置机制,将负载分发策略融入到流式处理系统中,并提供可用的切换机制;最后,基于 ABS (asynchronous barrier snapshot) [45] 算法实现了无中断键值状态交换机制,保证在动态负载均衡切换时不中断流式分析应用,保障系统的可用性.

4.2.1 负载分发策略

本文将负载均衡抽象为数据分配映射的目标优化,即$f: K \to N$,f将数据主键 k∈K 映射到 Flink 算子的子任务 n∈N 上,使得$\max C(n)/\min C(n)$最小化,其中$C: K \to Z$是子任务节点处理的数据量,从而提高负载均衡度,增加系统整体吞吐量.

典型流式分析应用场景中的数据分布可以由两个维度去判断,Key 分布的均匀性以及数据分布的均匀性.对于集群配置来说,也可以划分为资源均衡型和资源不均型.因此,本文根据常见的数据分布场景和集群配置情况的不同组合,分别提出 Modulo、LeastKey、LeastCount、Weight 四种负载分发策略,并支持在 KeyBy 算子上自定义负载分发策略.

(1) Modulo

当 KeyBy 所使用的 Key 为整数时,例如常见的数字 ID,且各 Key 的分布均匀(下称 Key 均匀分布)、各 Key 对应的数据量相当(下称数据均匀分布)时,负载分发可以通过 Key 对并行度取模,均衡地将负载分发到后续的算子的子任务上.

(2) LeastKey

当 KeyBy 所使用的 Key 不均匀分布,但数据均匀分布时,Modulo 策略将难以均衡分发负载.例如某市电压传感器的 ID 先以区级别分配前缀,如 A 区 100000、B 区 200000,后 5 位 ID 按各区内部安装顺序递增,此时由于 ID 的不连续性,以传感器 ID 进行 KeyBy 时,Modulo 策略可能导致较差的负载均衡.

针对这类场景,本文提出 Least 分配策略,Least 分配策略构造映射$f(k) = n$,其中$n \in N$且使得$S(n)$最小,$S: N \to Z$是节点的当前负载的度量函数.不同于批式处理中固定的输入数据集,由于流式处理系统需要处理无界的输入数据流,因此负载分发中仅以当前的负载状态作为依据.分布式集群下不同节点计算相同 Key 的目标子任务时,由于数据可能存在潜在乱序,系统的负载状态持续改变,导致计算产生的目标子任务状态发生改

变,因此 Least 策略的计算需要由特定的协调者进行计算和存储.

算法 4-1 Least 分发策略

Function: Least(Key, S)

Input: Key: Field used in KeyBy of data

S: Load function that measures the load of nodes

Output: Target node

```
1.  if Key is contained in History Assignment Map then
2.      return assigned node
3.  Select node N with minimize load by load function S
4.  Increase load of node N in load function S
5.  Assign N to Key in History Map
6.  return node N
```

Least 分发策略实现如算法 4-1 所示,Least 策略在协调器上执行,本地仅缓存分发结果.在 Least 分发策略中,如何选择具有最小负载的节点 N 是算法的关键部分,这取决于负载函数 S 如何存储节点的负载.本文使用哈希表存储节点对应的负载,使用跳表对节点负载进行排序,以实现常数时间复杂度的最小负载节点查找和对数时间复杂度的负载变更.

LeastKey 策略使用分配到节点上的 Key 数量作为衡量节点负载的函数 S,在分发负载时,选择当前负载最低的节点,因此适用于 Key 不均匀分布,但数据均匀分布的场景.

(3) LeastCount

LeastCount 和 LeastKey 的区别在于度量节点负载的 S 不同,当数据也不均匀分布时,需要根据 Key 对应的数据量进行负载分发,由于无界数据流的 Key 对应的数据量比例往往在一定时间内不会剧烈变化,可以使用历史的比例信息反映当前的状况.LeastCount 使用分配到节点上的 Key 对应的历史数据量之和对节点负载进行度量,并在分发负载时选择当前的负载最低节点,从而适用于数据不均匀分布场景,但需要历史数据的支撑.

(4) Weight

Flink 以 Task Slot 为单位调度计算任务,对所有的 Task Slot 一视同仁,每个 Task Slot 往往表示一个 CPU 核心.当 Flink 集群不同节点的 CPU 配置不同,导致单 CPU 核心性能存在差异时,即使不同节点的负载分布均衡,也会因为单核心性能差异导致吞吐量的差异,单核心性能孱弱的节点会成为系统的吞吐量瓶颈.本文提出带权重的负载分发策略,允许为每个节点定义不同的负载权重,从而在 Task Slot 间存在性能差异时,保证节点性能感知的负载均衡,该场景下的负载均衡度需要以节点性能进行加权计算.

各 Task Slot 按权重将[0, 100)划分为多个连续子区间,本文支持以 Key 的哈希值和随机数计算落点.当 Key 均匀分布时,本地基于哈希计算落点,从而选择目标算子子任务即可,当 Key 不均匀分布时,需要由特定的协调者基于随机数进行计算,保证负载的均衡.

Weight 分发策略的实现如算法 4-2 所示,使用哈希的 Weight 策略只需在本地计算,使用随机数的 Weight 策略在协调器上执行.在 Weight 分发策略中,对于节点数量较少的集群,朴素的遍历足以满足性能,当集群节点数量较多时,本文通过将 WeightTable 预处理为累计和,例如{20, 50, 30}的 WeightTable 会被预处理为{20, 70, 100},从而可以采用二分查找代替 4-6 行以实现对数时间复杂度的落点计算.

算法 4-2 Weight 分发策略

Function: Weight(Key, WeightTable)

Input: Key: Field used in KeyBy of data

WeightTable: Weight of nodes, sum of weights should equal to 100

```
Output: Target node
1.  if Key is contained in History Assignment Map then
2.      return assigned node
3.  Initiate R as a random number in [0, 100), N as node0
4.  while (R >= WeightTable[N]) do
5.      R = R - WeightTable[N]
6.      N = next node
7.  Assign N to Key in History Map
8.  return node N
```

综上,表 5 分发策略及适用场景总结了包括本文提出的四种分发策略在内的负载分发策略及其所适用的场景.由于实时数据流具有随机性,相较于传统的数据倾斜问题,KeyBy 算子负载分发更关注于数据流的实时分布特性以及历史状态信息.为了应对动态变化的流数据,在 4.2.2 节和 4.2.3 节中,本文进一步提出了基于在线监测的负载均衡策略动态配置和基于 ABS 算法的无中断键值状态交换.

表 5 分发策略及适用场景

分发策略	分配映射方式	本地计算	适用场景
Hash	基于哈希函数与键值组	是	Key 不均匀分布,数据不均匀分布,节点性能不明
Modulo	对并行度取模	是	Key 为整数、Key 均匀分布、数据均匀分布
LeastKey	分配到负载最低节点 负载通过 Key 数量计算	否	数据均匀分布
LeastCount	分配到负载最低节点 负载通过历史数据量计算	否	数据不均匀分布
Weight	根据节点权重分配	取决于配置	节点间硬件配置不同

4.2.2 基于在线监测的负载均衡策略动态配置

为了衡量比较不同分配策略的优劣,需要在线检测不同策略的负载均衡度,本文通过对 KeyBy 算子进行扩展,支持按可配置的数据采样率,对流式数据进行采样后,计算在所有支持的分发策略下的负载均衡度.数据流经 KeyBy 算子时,采样线程根据配置的采样率对数据随机采样,计算后自动化配置最优负载分发策略.

更高的采样率意味着对数据流更准确的反映,但也可能导致更多的计算代价.在流式处理应用刚启动时,系统可能缺少数据分布情况的信息,只能使用默认的负载分发策略,并选择高采样率.在一段时间的观测后,可以根据监测信息,选择更优的分发策略和更低的采样率,此时不得不中断应用的执行,调整配置后重新启动,这会造成短时间的不可用和带来复杂的运维工作.针对这一问题,本文基于消息发布订阅机制,支持在任务运行时动态调整配置,包括采样率、KeyBy 负载分发策略.当配置修改提交至动态配置中心提时,动态配置中心基于消息发布/订阅通道将指令下发到 KeyBy 算子配置管理模块上,从而控制 KeyBy 算子.

基于在线监测与动态配置调整,本文通过运行时自动调优分发策略实现动态负载均衡,通过对采样数据的汇总分析,系统可以自动选择最优的 KeyBy 分配策略并进行动态调整,实现系统的负载均衡优化.动态负载均衡支持定期、定量、阈值三种模式: 1) 定期模式由定时器触发后切换到当前具有最优负载均衡度的负载分发策略,适合数据源流速稳定的场景;2) 定量模式在接收到指定数量数据时触发,适合数据源流速变化的场景;3) 阈值模式则在当前选用策略的负载均衡度低于配置阈值时触发,适用于用户希望尽可能减少策略切换的场景.

这三种模式总结如表 6 动态负载策略触发条件及场景所示.

表 6 动态负载策略触发条件及场景

模式	触发条件	适用场景
定期	定时	数据源流速稳定
定量	接收到指定数量数据	数据源流速不稳定
阈值	负载均衡度低于阈值	希望尽可能减少策略切换

4.2.3 基于 ABS 算法的无中断键值状态交换

调整 KeyBy 算子负载分发策略时,由于分配映射关系被改变,需要交换各节点持有的 Keyed State,因此需要中断流式分析应用的执行,并通过 Savepoint 完成状态的交换,这会导致服务长时间不可用和数据积压.

典型流式系统使用 Checkpoint 提供容错和故障恢复机制,Flink 的 Checkpoint 生成采用 Chandy-Lamport 算法[46]的变种——ABS(asynchronous barrier snapshot)算法.其中,作业可以抽象成图,图的顶点是算子,边是数据流,与 Chandy-Lamport 算法的图模型对应.ABS 算法通过 Barrier 将数据流从时间上划分开,Barrier 作为快照的边界,用于区分当前快照的数据和下一次快照的数据.Barrier 从 Source 算子沿数据流流动,当一个算子接收到所有输入数据流的 Barrier 时,生成状态快照,并继续向下游算子广播 Barrier,快照生成后,解除对数据流的阻塞,继续计算,如图 10 基于 ABS 算法的无中断交换示意图.

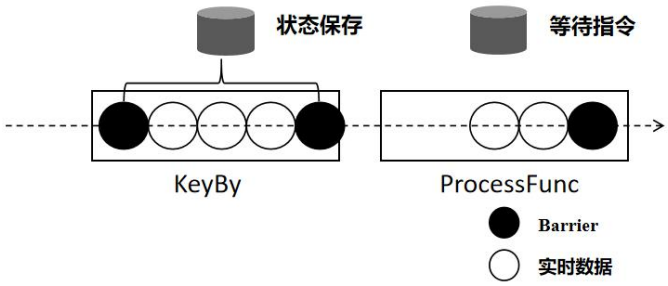

图 10 基于 ABS 算法的无中断交换示意图

为了实现在不中断流式分析应用的情况下完成键值状态交换,本文提出基于 ABS 算法的 Sync 指令,用于触发存储键值状态的状态后端在节点间的状态交换,从而在不中断应用情况下完成 KeyBy 算子负载分发策略的动态调整.当 Source 算子接收到 Sync 指令时,会向 KeyBy 算子传递特殊的屏障标记并阻塞当前通道的输入;当 KeyBy 算子接收到来自所有数据通道的标记时,存储状态的状态后端生成状态快照,并解除所有数据通道的阻塞.在数据流上,本文将 KeyBy 及其后的有状态处理函数视为 ABS 算法中的一个节点,当节点的通道接收到屏障标记时,该通道被阻塞,当所有通道都接收到屏障标记时,生成键值状态的快照,解除所有通道的阻塞.

4.3 基于键值的数据交换策略

本文提出 Local Merge、Global Merge 和 Ahead KeyBy 三种基于键值的数据交换策略,通过减少或避免 KeyBy 的全量数据传输,以提高吞吐量和降低传输延迟.

(1) Local Merge

Flink 的编程模型在 MapReduce 编程模型上进行扩展,也通过并行执行算子任务支持任务并行,且支持更多样化的算子.其 KeyBy 算子类似于 MapReduce 中的 Shuffle 阶段,但不支持 Combine 函数进行本地数据聚合.Flink 仅在 Flink SQL 中支持 Local-Global Aggregation[47],其目标是解决数据倾斜问题,不适用于事件驱动型流式分析应用.流式处理中,由于数据具有时序信息,不能简单地进行聚合.本文提出 Local Merge 算子,支持时序

感知的本地数据聚合,从而减少 KeyBy 时的网络传输数据量.

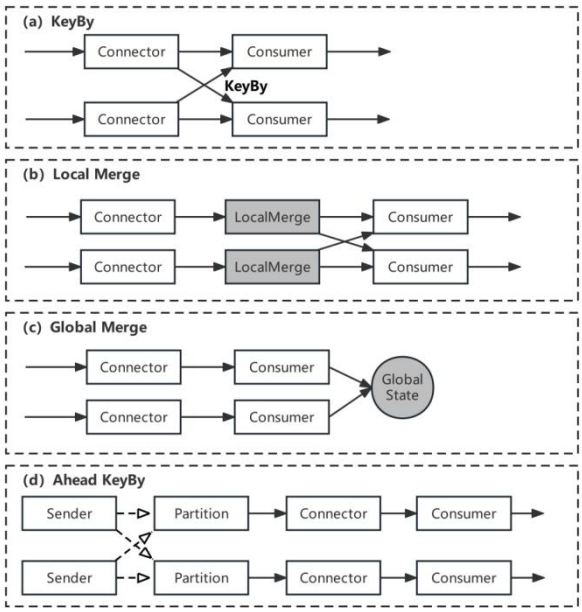

图 11 基于键值的数据交换策略

Local Merge 算子将事件划分到不同的时间槽中分别聚合,聚合后事件的时间戳被设置为所有事件的最大值,以向后传递正确的水位线,如图 11 基于键值的数据交换策略 b)所示.Local Merge 算子在向后发射数据时,会按时间槽向后顺序依次发送,保持数据有序.Local Merge 算子支持定期、定量和定点的数据发射模式,定期模式在最早被聚合数据到达指定延迟时发射,适合要求低处理延迟场景;定量模式在聚合指定数量的数据后发射,适合要求高吞吐量场景;定点模式识别到特定事件时发射,可以在定期和定量模式基础上,用于检测到窗口边界后发射数据,降低结果的产生延迟;如表 7 Local Merge 模式触发条件及适用场景所示.

表 7 Local Merge 模式触发条件及适用场景

模式	触发条件	适用场景
定期	到达指定延迟	低延迟
定量	聚合指定数量数据	高吞吐
定点	数据满足定点条件	检测窗口边界等

(2) Global Merge

在低电压检测应用场景中,大部分实时电压数据属于正常电压,不会修改当前全局电压状态,仅需要进行本地聚合.这类计算任务在异常监管等场景中较为普遍,大量局部计算并不会修改全局状态.因此在各节点单独计算,将部分结果存储在本地状态中,并将状态变更聚合到全局状态中能有效节约通信成本.全局状态可以使用如键值存储 Redis、关系型存储 MySQL[48]等实现.

Global Merge 是一种基于键值的数据交换策略,包括本地状态存储和全局状态存储,Global Merge 不在节点间交换数据,避免 KeyBy 带来的网络传输,基于键值的分析结果在各节点单独计算,通过结果聚合函数最终聚合到全局状态管理器上,如图 11 基于键值的数据交换策略 c)所示.Global Merge 策略适用于数据对结果的修改较少或结果延迟要求较宽松的场景,当结果延迟要求宽松时,结果的修改可以先缓存在本地进行聚合,定期

或定量地与全局结果聚合,这使得最终的聚合状态请求量远小于数据量,从而减少网络传输.本地状态与全局状态的聚合也支持定期、定量和定点模式.

(3) Ahead Keyby

当流式处理系统使用分布式消息队列作为数据源时,由于这些分布式消息队列将数据划分到多个分区进行存储,划分过程可以是基于轮转、哈希或者主键的,因此 KeyBy 操作可以被提前到消息队列上进行,如图 11 基于键值的数据交换策略 d)所示.因为数据从数据源发送到消息队列的网络传输成本是固定存在的,Ahead KeyBy 通过提前在消息队列进行 KeyBy,可以在不增加额外成本的情况下,避免后续流式计算中使用 KeyBy 算子进行数据交换,而一对一的算子间数据传输会发生在同一计算节点的内存上,从而可以避免网络数据传输.Ahead KeyBy 将数据划分到不同的分区时,与 KeyBy 算子将数据分发到不同的子任务节点上类似,也因此可以受益于上述的负载分发策略.

本文提出 Local Merge、Global Merge 和 Ahead KeyBy 三种基于键值的数据交换策略,适用于不同的流式数据分布场景,以供用户使用.

5 系统实现与整体架构

本工作以原生 Flink 框架为基础,实现并扩展了上述三个方面的性能优化策略.总体而言,本文从数据流的角度出发,提出了 KeyBy 算子负载分发策略及其扩展、键级水位线机制、基于键值的数据交换策略,与具体的处理算法和计算过程解耦,形成了一套通用的高性能流式处理优化数据流,如下图 12 Trilink 数据流图所示;通过对原生 Flink 系统架构扩展,实现了原型系统 Trilink.

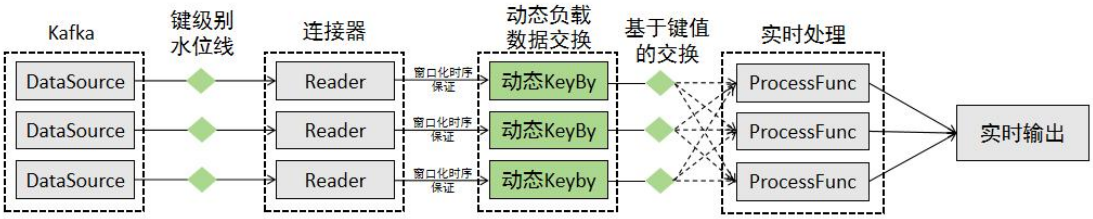

图 12 Trilink 数据流图

Trilink 基于 Filnk 实现,其原型架构如下图 13 Trilink 系统架构图所示,其中的扩展部分主要包括:

1) 在键级水位线实现中,由于 Flink 提供了 KeyedProcessFunction 接口对 KeyedStream 进行处理,KeyedProcess Function 接口保障其处理的全部数据,拥有相同的 key K.本文据此抽象出 KeyedWatermarkProcessFunction 类,包含窗口更新,水位线更新等关键方法.

2) KeyBy 算子负载分发策略与动态负载均衡的设计实现主要包括:位于 TaskManager 的 KeyBy 负载分发策略、数据采样、Sync 状态后端、配置管理中心模块,以及位于 JobManager 中的 KeyBy 策略协调器、数据汇总与动态负载均衡、Sync 协调器、动态配置管理中心模块;

3)基于键值的数据交换策略包括 Local Merge 和 Global Merge,其中 Local Merge 通过 TaskManager 中的 Local Merge 算子实现,Global Merge 通过位于 TaskManager 中的本地状态和位于 JobManager 中的全局状态管理器实现.

Flink编程片段:

```
DataStream<Event> stream = ...;
DataStream<MappedResult> mappedStream =stream
   .keyBy(event ->event.getKey())
   .window(TumblingEventTimeWindows.of(Time.minutes(5)))
   .map(new MapFunction<Event, MappedResult>() {
            ...
});
```

Trilink编程片段:

```
DataStream<Event> stream = ...;
DataStream<MappedResult> balancedStream = stream.
   DynamicBalance(thresholdTrigger, interval);
DataStream<MappedResult> mappedStream = balancedStream
   .KeyedWatermark(event -> event.getKey(), Time.minutes(5), new MapFunction<Event, MappedResult>() {
            ...
});
```

图 13 Trilink 编程片段示意

对于扩展的模块,系统提供配置项和算子,用户可以通过使用相应的算子和配置便捷地组成所需的高吞吐流式处理数据流.图 13 是一个具体的用户编程片段,可以看到,相较于 Flink 框架,Trilink 只引入了部分额外算子及配置.

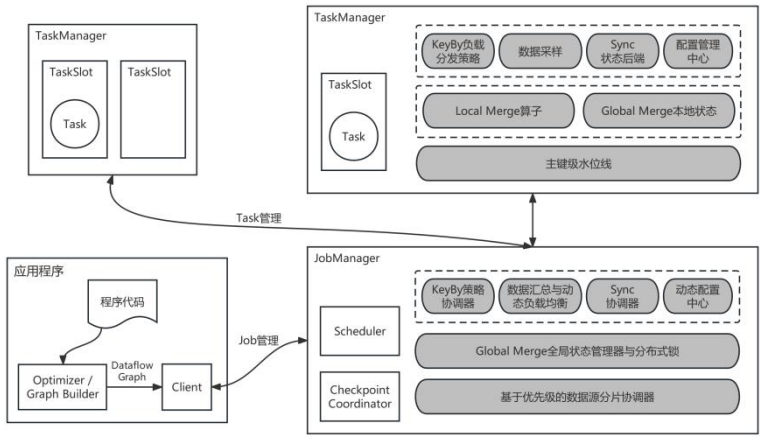

图 14 Trilink 系统架构图

6 系统实验与评价

6.1 实验目的

实验主要目的在于分析评估本文提出的优化策略和技术的有效性,同时分析评估介绍对整体应用吞吐量的影响,以及与 Flink 原有机制或策略的性能对比.

本章实验评价将回答以下研究问题:

问题 1（RQ1）优化后的 Trilink 系统与原生 Flink 相比,性能是否有所提升,可扩展性是否有所提高?

问题 2（RQ2）Trilink 中所设计的三种优化机制在性能上分别有什么样的效果?

6.2 实验环境与实验方法

本文所使用的实验环境为 8 台机器所组成的集群,每台机器具有一个型号为 Intel(R) Xeon(R) Gold 5215 @ 2.50GHz 的 CPU 和 128GB 的内存,机器间通过路由器连接,网络为千兆网络.8 台机器均部署 Flink worker、Zookeeper、Kafka,其中一台作为集群 Master,部署 Flink master 和 Redis,所使用的操作系统和软件版本如表 8 实验环境软件版本所示.

表 8 实验环境软件版本

软件	软件版本
OS	CentOS Linux release 8.0.1905
JDK	1.8.0_301
ZooKeeper	3.7.0
Kafka	3.1.0
Redis	6.2.6
Flink	1.14.0

本文选用低电压检测应用,桥梁拱顶监测数据集和 Yahoo Streaming Benchmark 作为测试负载,重点分析评估 Trilink 相比于 Flink 和 GeoFlink 在吞吐量和水平扩展加速比的提升效果.测试数据来源于真实历史电压数据,交通基础设施建设传感器数据和数据生成器模拟的业务数据.其中,桥梁拱顶监测场景下,需要以低时延,高频率对桥梁施工建设状态进行分析从而确保基础设施建设可靠性.测试数据预先储存在 Kafka 中,并在数据首尾做特殊的标记以精确地测量数据处理时间,从而计算应用吞吐量.

由于 Kafka 支持重放数据,优化前后的应用都在相同的数据集上进行测试.测试中先对应用进行预热,以消除文件缓存等的影响,所有测试结果采用三次测量的平均值,以减小实验误差.

本文也基于低电压检测应用及相关电压数据,对提出的三种优化技术进行单独测试,关注其分别对于性能提升的影响.在 KeyBy 算子分发策略中,本文关注不同策略在不同流式数据分布情况下的负载均衡度,以及负载不均衡对应用整体吞吐量的影响.在键级水位线和基于键值的数据交换策略中,本文关注应用的整体吞吐量.

6.3 系统总体性能优化测试与评估

本文在低电压检测应用上应用所提出的三种优化技术,将数据分布和集群配置与各策略适用场景进行结合.具体来说,与优化前相比,使用键级水位线代替子任务级水位线,初始默认使用 LeastKey 负载分发策略,在线监测参数设置为阈值,数据交换策略选择 Ahead Keyby,在桥梁拱顶监测数据集和 Yahoo Streaming Benchmark 中,为了保持测试一致性,本文也进行类似的优化.

实验结果如图 15 所示,相较于 Flink,低电压检测应用每秒吞吐量从单机 42.5 万提升到 560 万,且由单节点扩展到八节点时,吞吐量提升到 6.56 倍,达到理想情况下 8 倍吞吐量的 82%.相比于原来的 1.24 倍,水平可扩展性得到明显改善,提升了 4.3 倍.拱顶监测应用每秒吞吐量从单机 37.8 万提升到 510 万,且由单节点扩展到八节点时,吞吐量提升到 5.4 倍,达到理想情况下 8 倍吞吐量的 68%.相比于原来的 2.1 倍,水平可扩展性得到明显改善,提升了 1.6 倍.Yahoo Streaming Benchmark 的每秒吞吐量从单机 71.9 万提升到 430.6 万.且由单节点扩展到八节点时,吞吐量提升到 6.4 倍,达到理想情况下 8 倍吞吐量的 80%.相比于原来的 2 倍,水平可扩展性得到明显改善,提升了 2.2 倍.相较于 GeoFlink,三种应用下的单机吞吐率和水平扩展加速比 Trilink 均体现出显著优势.需要注意的是,图中横坐标分布并非均匀,随着节点数量增加吞吐量的增长趋势呈亚线性趋势.

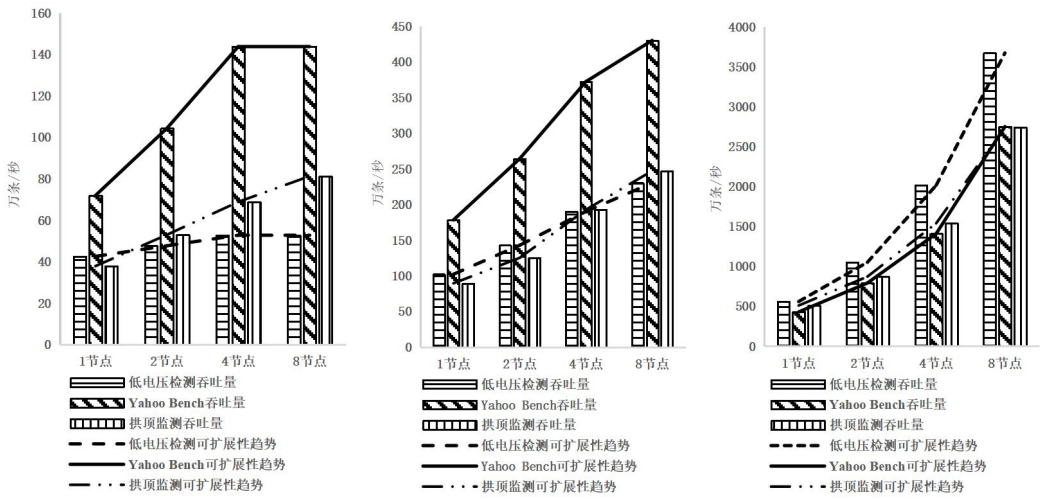

图 15 Flink(左),GeoFlink(中)与 Trilink(右)吞吐量与可扩展性实验

整体而言,Trilink 相比于 Flink 显著提供了流式数据处理性能.即使在单机情况下,Trilink 的吞吐能力仍然优于原生 Flink.另一方面,从水平可扩展性的角度来看,本文提出的优化设计充分考虑了流式处理通用数据流的三个性能瓶颈之处,所以系统呈现了更优的系统可扩展性.

需要注意的是,采用 Trilink 后,低电压检测应用的系统的性能可扩展性提升更为显著,其原因在于,低电压检测应用中数据源更为庞杂,原生 Flink 系统水位线机制造成了巨大的额外时序耦合,从而导致系统吞吐率和可扩展性显著下降.另一方面,KeyBy 动态负载策略也为这种复杂的数据分布带来了更高效的分发处理能力.

6.4 不同优化策略效果的测试与评估

该实验基于低电压检测应用及相关电压数据,对提出的三种优化技术进行单独应用和测试,以分析和评估每个优化策略对于系统性能提升的效果.相较于 Yahoo Benchmark,低电压检测应用的数据流更通用基础,且来源于真实应用,在所有流式处理应用中均能找到与其流程相符的部分,因而,用这一数据作为测试负载更具备通用性.在 KeyBy 算子分发策略中,本文关注不同策略在不同流式数据分布情况下的负载均衡度,以及负载不均衡对应用整体吞吐量的影响.在键级水位线和基于键值的数据交换策略中,本文关注应用的整体吞吐量.

6.4.1 KeyBy 负载分配策略测试与评估

基于在线监测功能,本文对四个主要的 KeyBy 算子分配策略进行负载均衡度测试.主要测试四种流式数据分布场景,包括 Key 均匀分布且每个 Key 的数据量也均匀分布（下称数据均匀分布）、Key 非均匀分布但数据均匀分布、Key 均匀分布但数据非均匀分布、Key 非均匀分布且数据非均匀分布.

本实验延用第 3 节中提出的负载均衡度指标.实验结果如图 16 KeyBy 负载分发策略与负载均衡测试所示,Flink 原有的基于 Hash 的方式在四种场景下均表现不佳,最低的负载均衡度仅有 0.26,这导致 70%的额外计算时间.Modulo 策略在 Key 均匀分布时负载均衡度接近 1,但在后三个场景降低至 0.66、0.41 和 0.24.LeastKey 策略在 Key 非均匀分布时仍具有接近 1 的负载均衡度,但不适用数据非均匀分布场景.LeastCount 策略在缺少历史分布信息的情况下,与 LeastKey 没有显著差异,在有历史信息支撑的情况下,在三种场景至少都有 0.97 的负载均衡度.

虽然 Least 策略表现总比 Hash 或者 Modulo 策略优秀,但存在需要跟中心分配服务通信的开销,因此在冷启动时会对性能有影响.当 Hash 或者 Modulo 满足使用者要求时,不需要使用 Least 策略.LeastCount 需要历史信息的支持,在系统冷启动时与 LeastKey 没有显著差异,因此对于数据均匀分布场景选择 LeastKey 即可.

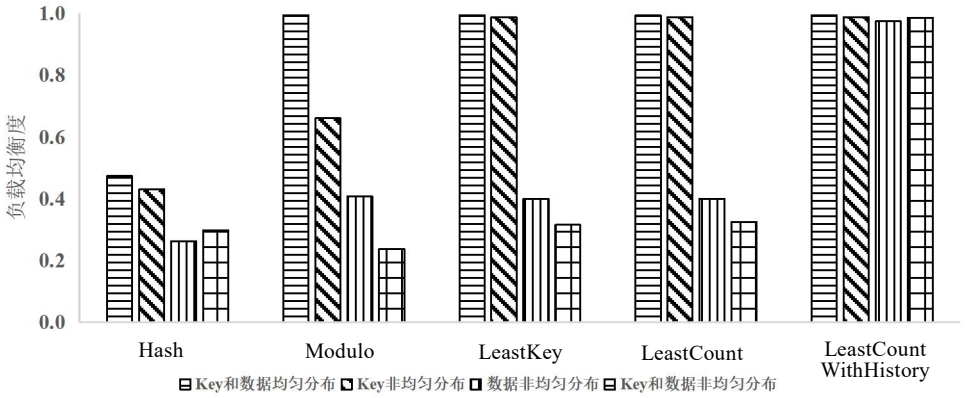

图 16 KeyBy 负载分发策略与负载均衡测试

进一步的,本文测试了 Hash 和四种算子分发策略下,流式处理系统吞吐率.结果如图 17 所示,在 Key 和数据均匀分布场景下,modulo 策略下由于负载分配均衡,吞吐量提升到基准性能的 3 倍以上.在其余 3 个场景下,相较于 Hash 策略,四种策略均体现出了显著性能优势.

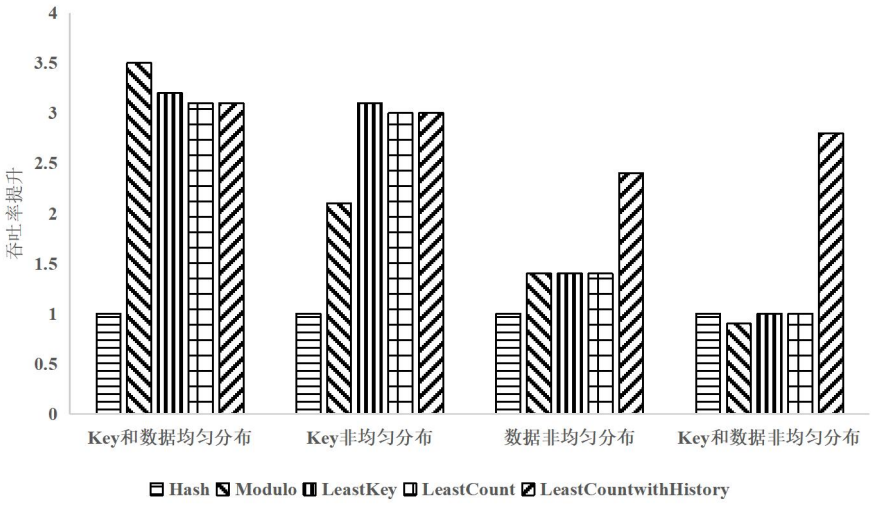

图 17 负载分发策略吞吐量提升示意图

在 4.2 节中,本文提出了基于在线监测的负载均衡动态配置方法,该方法支持定量、定期、阈值三种模式,三种模式适用于不同的场景.我们定义数据流速变化常数 a,数据流速相对上一时刻变化率大于 a 即代表此时发生数据分布变化.数据流速变化速度 v,v 代表单位时间(分钟)内数据流速变化次数.下面我们定义 a = 0.5,初始策略为 Modulo,观察不同数据分布变化速度下,三种参数对系统性能产生的影响,如下表 9 所示.实验结果发现,定期模式适合数据源流速相对稳定的场景;定量模式适合数据源流速变化相对频繁的场景;阈值模式适用于尽可能减少策略切换的场景.最优参数选择需要考虑数据流速和数据流速变化频繁程度.

表 9 不同流速变化下在线监测参数对系统吞吐量(万条)的影响

流速变化速度 (次/分钟)	0.1	1	5	10	30
定量(1 亿条)	130.1	128.4	76.2	61.8	47.8
定量(1000 万条)	101.2	97.2	72.7	70.5	68.1
定期(20 秒)	96.1	95.7	74.9	70.1	67.2
定期(5 秒)	82.2	83.1	81.9	80.3	75.4
阈值(0.6)	131.1	129.2	90.1	82.1	64.2
阈值(0.4)	130.2	128.7	85.4	89.2	78.8

6.4.2 键级水位线测试与评估

本文选用低电压检测应用,对子任务级水位线和主键级水位线的乱序数据比例和性能差异进行测试.由于各电压传感器数据分别有序,该场景在主键级水位线下没有乱序数据,但在子任务级水位线下,由于网络时延和物理时钟的影响,往往会存在时序耦合的现象.为了保证结果的正确性,需要基于迟到数据更新结果,这些额外的更新操作会导致系统吞吐量的下降.实验分别在数据乱序程度为 25%,50%,75%的情况下进行测试,主键级水位线下的应用吞吐量,是子任务级水位线下的 5 倍以上,如表 10 子任务级与键级水位线数据乱序性能测试所示.

表 10 子任务级与键级水位线数据乱序性能测试

数据乱序比例	25%	50%	75%
Flink 子任务级（万条/秒）	42.5	28.2	13.6
Trilink 键级（万条/秒）	244.3	241.8	228.5

在现实低电压检测真实业务中,往往要求秒级结果输出,由于地理位置和网络通信的影响,实时数据由于子任务级水位线导致的时序耦合非常严重,平均在 50%以上.使用 Flink 原生子任务级水位线会导致系统实时性能的大幅度下降.而本文的提出的键级水位线优化机制能够有效避免这一问题.

6.4.3 基于键值的数据交换策略测试与评估

实验测试不同的基于键值的数据交换策略下的吞吐量和网络传输的数据量.测试结果表明,Local Merge 策略仅在网路中传输 Flink 原有策略传输数据量的 1.49%,而 Global Merge 策略和 Ahead KeyBy 策略不需要在网络中传输数据.通过减少或避免在网络中进行基于键值的数据交换,可以在网络带宽受限环境中,将吞吐量提高到 3.92 倍以上,如下图 18 数据交换策略性能测试所示.

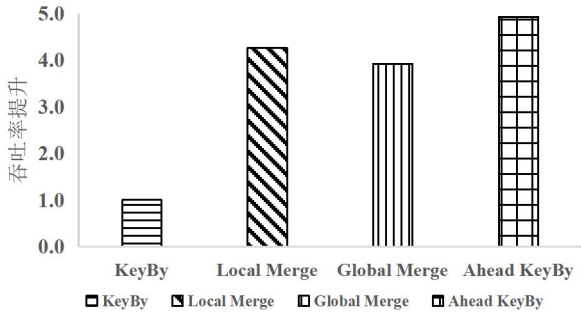

图 18 数据交换策略性能测试

6.5 实验讨论

通过实验评估,本研究提出的三种优化策略显著提高了流式处理系统的性能. 相较于 Flink,Trilink 单机吞吐量提升 6 倍以上,八节点集群下水平扩展加速比提升 2.2 倍以上,呈现出亚线性增长趋势.

本研究针对数据流额外引入了细粒度的水位线,同时也带来额外的系统开销.相较于 Flink 中的子任务级水位线机制,主要带来的额外开销包括:水位线存储空间开销,状态定时器空间开销以及水位线计算更新开销.在空间开销上,当键数量为 n 时,对于每个键,系统需要维护一个水位线的状态,通常是一个时间戳.每个水位线占用 8 字节（64 位时间戳）.如果系统为每个键维护一个独立的定时器,具体的内存占用取决于定时器对象的大小,定时器空间占用约为几十字节.此外,在流计算的场景中,数据经过使用后会被丢弃,因此当该键不再产生数据后,这一水位线会被自动释放,不会造成额外的空间占用.在大部分场景中,我们认为这一开销是可以接受的.对于计算开销,当事件处理时,需要更新相应键的水位线,涉及到查找水位线、比较时间戳、写入操作等,其开销取决于键分布情况,数据动态变化情况以及具体计算任务.因此,在 Trilink 系统中,同样支持 N-键级水位线机制,即 N 个 Key 共享同一水位线,从而达到空间开销与系统性能间的均衡.键级水位线机制实际提供了一种细粒度的控制方式,用户可以在键级和子任务级间自主调控,从而达到高性能高可用.

另一方面,在进行 KeyBy 分发策略在线监测与切换时,需要对 Key 状态进行迁移,存在一定的开销.因此本文为在线切换提供了三种不同策略,用户可以根据数据流分布特性与变化规律选择合适的参数,从而降低状态切换带来的额外开销.在 6.4.1 节实验测试中,本文分别用三种模式引入了状态调度切换后,系统性能相较于 Flink,仍然有显著提高.

尽管目前的键级水位线支持细粒度的控制,但是不支持自动的优化配置与增量学习,未来工作中可以采用机器学习或规则约束方法对流式数据分布情况进行预测以智能化地增量调整水位线粒度.同时,当前 KeyBy 负载分发策略中涉及到多节点并行计算,可能存在潜在的一致性问题,后续工作将使用高效的分布式一致性协议解决 KeyBy 负载分发策略中多节点计算的一致性问题.

7 总结

本文总结分析典型的事件驱动型流式分析应用的数据流特征,并通过测量实验分析并总结这类应用存在的水平可扩展性瓶颈问题,最后针对性的从水位线设置、负载均衡和跨节点数据交换几个方面提出了优化策略.基于上述优化技术,论文通过扩展现有的原生 Flink 框架实现了原型系统 Trilink.实验结果表明,相比于 Flink,Trilink 对于真实应用和测试基准的系统吞吐率提升 6 倍以上,同时系统的水平扩展加速比得到 1.6 倍以上的显著提高,呈现亚线性增长趋势.

本文针对的流处理负载是较为通用的流式分析应用,目前基于 Apache Flink 流式处理系统实现,该方法同样适用于其他分布式流式处理框架,在后续工作中,本文工作者会在其他框架上实现原型系统.

未来工作方面,当前 Trilink 的键级水位线设计在应对大多数场景时已显示出性能优势,我们计划引入机器学习算法,根据历史数据流特征动态调整水位线粒度策略,提高水位线的精细化设置.除此之外,计划开发基于负载预测的动态资源调度算法,实时监控系统负载并预测未来负载变化,自动调整计算资源的分配和回收.通过实现资源的弹性伸缩,确保系统在负载高峰时能够快速响应,同时在负载减少时有效节约资源.

References:

[1] Manyika J, Chui M, Brown B, et al. Big data: The next frontier for innovation, competition, and productivity[M]. McKinsey Global Institute, 2011.

[2] Yaqoob I, Hashem I A T, Gani A, et al. Big data: From beginning to future[J]. International Journal of Information Management, 2016, 36(6): 1231-1247.

[4] Smutny P, Schreiberova P. Chatbots for learning: A review of educational chatbots for the Facebook Messenger[J]. Computers & Education, 2020, 151: 103862.

[5] Cao W, Gao Y, Li F, et al. Timon: A timestamped event database for efficient telemetry data processing and analytics[C]//Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2020: 739-753.

[6] Apache Flink[EB/OL].[2024-01-19]. https://flink.apache.org/

[7] Zaharia M, Chowdhury M, Franklin M J, et al. Spark: Cluster computing with working sets[C]//2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10). 2010.

[8] Fragkoulis M, Carbone P, Kalavri V, et al. A survey on the evolution of stream processing systems[J]. The VLDB Journal, 2023: 1-35.

[11] Zeuch S, Monte B D, Karimov J, et al. Analyzing efficient stream processing on modern hardware[J]. Proceedings of the VLDB Endowment, 2019, 12(5): 516-530.

[12] Chintapalli S, Dagit D, Evans B, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming[C]//2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW). IEEE, 2016: 1789-1792.

[13] Terry D, Goldberg D, Nichols D, et al. Continuous queries over append-only databases[J]. Acm Sigmod Record, 1992, 21(2): 321-330.

[14] Javed M H, Lu X, Panda D K. Characterization of Big Data Stream Processing Pipeline: A Case Study using Flink and Kafka[C]//Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies. ACM, 2017: 1-10.

[15] Chandrasekaran S, Cooper O, Deshpande A, et al. TelegraphCQ: continuous dataflow processing[C]//Proceedings of the 2003 ACM SIGMOD international conference on Management of data. ACM, 2003: 668-668.

[16] Gedik B, Andrade H, Wu K L, et al. SPADE: the system s declarative stream processing engine[C]//Proceedings of the 2008 ACM SIGMOD international conference on Management of data. ACM, 2008: 1123-1134.

[17] Arasu A, Babcock B, Babu S, et al. STREAM: The Stanford Data Stream Management System[M]//Data Stream Management. Springer Berlin Heidelberg, 2016:271 - 275.

[18] Abadi D J, Carney D, Çetintemel U, et al. Aurora: a new model and architecture for data stream management[J]. the VLDB Journal, 2003, 12(2): 120-139.

[19] Abadi D J, Ahmad Y, Balazinska M, et al. The Design of the Borealis Stream Processing Engine[C]//Cidr. 2005, 5(2005): 277-289.

[20] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[21] Apache Storm[EB/OL]. [2023-11-22]. https://storm.apache.org/

[22] Armbrust M, Das T, Torres J, et al. Structured streaming: A declarative api for real-time applications in apache spark[C]//Proceedings of the 2018 International Conference on Management of Data. 2018: 601-613.

[23] Friedman E, Tzoumas K. Introduction to Apache Flink: Stream Processing for Real Time and Beyond[M]. " O'Reilly Media, Inc.", 2016.

[24] Zeuch S, Monte B D, Karimov J, et al. Analyzing efficient stream processing on modern hardware[J]. Proceedings of the VLDB Endowment, 2019, 12(5): 516-530.

[25] Koliousis A, Weidlich M, Castro Fernandez R, et al. Saber: Window-based hybrid stream processing for heterogeneous architectures[C]//Proceedings of the 2016 International Conference on Management of Data. 2016: 555-569.

[26] Mai L, Zeng K, Potharaju R, et al. Chi: A scalable and programmable control plane for distributed stream processing systems[J]. Proceedings of the VLDB Endowment, 2018, 11(10): 1303-1316.

[27] Varga B, Balassi M, Kiss A. Towards autoscaling of Apache Flink jobs[J]. Acta Universitatis Sapientiae, Informatica, 2021, 13(1): 39-59.

[28] Arkian H R, Pierre G, Tordsson J, et al. Model-based stream processing auto-scaling in geo-distributed environments[C]//2021 International Conference on Computer Communications and Networks (ICCCN). IEEE, 2021: 1-10.

[29] He C, Huang Y, Wang C, et al. Dynamic Data Partitioning Strategy Based on Heterogeneous Flink Cluster[C]//2022 5th International Conference on Artificial Intelligence and Big Data (ICAIBD). IEEE, 2022: 355-360.

[30] Tangwongsan K, Hirzel M, Schneider S. Optimal and general out-of-order sliding-window aggregation[J]. Proceedings of the VLDB Endowment, 2019, 12(10): 1167-1180.

[31] Shahvarani A, Jacobsen H A. Parallel index-based stream join on a multicore cpu[C]//Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2020: 2523-2537.

[32] Karimov J, Rabl T, Markl V. Astream: Ad-hoc shared stream processing[C] //Proceedings of the 2019 International Conference on Management of Data. 2019: 607-622.

[33] Karimov J, Rabl T, Markl V. AJoin: ad-hoc stream joins at scale[J]. Proceedings of the VLDB Endowment, 2019, 13(4): 435-448.

[34] McSherry F, Lattuada A, Schwarzkopf M, et al. Shared arrangements: practical inter-query sharing for streaming dataflows[J]. arXiv preprint arXiv:1812.02639, 2018.

[35] Zhang X, Ma K. Toward Sliding Time Window of Low Watermark to Detect Delayed Stream Arrival[C]//16th EAI International Conference. 2020: 444-454.

[37] Shaikh S A, Mariam K, Kitagawa H, et al. GeoFlink: A distributed and scalable framework for the real-time processing of spatial streams[C]//Proceedings of the 29th ACM International Conference on Information & Knowledge Management. 2020: 3149-3156.

[38] Putatunda S, Laha A K. Travel Time Prediction in Real time for GPS Taxi Data Streams and its Applications to Travel Safety[J]. Human-Centric Intelligent Systems, 2023: 1-21.

[39] Apache Kafka[EB/OL]. [2023-12-07]. https://kafka.apache.org/

[40] Redis [EB/OL]. [2024-01-09]. https://redis.io/

[41] Akidau T, Begoli E, Chernyak S, et al. Watermarks in stream processing systems: Semantics and comparative analysis of apache flink and google cloud dataflow[R]. Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2021.

[42] Wilmanns P S, Geuns S J, Hausmans J P H M, et al. Buffer sizing to reduce interference and increase throughput of real-time stream processing applications[C]//2015 IEEE 18th International Symposium on Real-Time Distributed Computing. IEEE, 2015: 9-18.

[43] Gulisano V, Palyvos-Giannas D, Havers B, et al. The role of event-time order in data streaming analysis[C]//Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems. 2020: 214-217.

[44] Saxena P. Analysis of Various Hash Function[J].

[45] Aumayr D, Marr S, Gonzalez Boix E, et al. Asynchronous snapshots of actor systems for latency-sensitive applications[C]//Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes. 2019: 157-171.

[46] Chandy K M, Lamport L. Distributed snapshots: Determining global states of distributed systems[J]. ACM Transactions on Computer Systems (TOCS), 1985, 3(1): 63-75.

[47] Performance Tuning[EB/OL]. [2023-03-01]. https://nightlies.apache.org/flink/ flink-docs-release-1.14/docs/dev/table/tuning/.

[48] MySQL[EB/OL]. [2023-03-01]. https://www.mysql.com/

附中文参考文献:

[3] 傅宣登,吴志林.Apache Flink 复杂事件处理语言的形式语义.软件学报,,():1-23

[9] 曾鸿斌.面向事件驱动型流式分析应用的 Flink 框架性能优化[D].北京:中国科学院大学,2023.

[10] 徐志榛,徐辰,丁光耀,陈梓浩,周傲英.支持实时流计算应用的关键技术研究进展.软件学报,2024,35(1):430-454

[36] 岳晓飞,史岚,赵宇海,季航旭,王国仁.面向 Flink 迭代作业的动态资源分配策略.软件学报,2022,33(3):985-1004