# 适应元组分布倾斜的分布式日志流处理系统

赵静芬[1,2]，张鹏[1,2]，孙永[1,2]，刘庆云[1,2]

(1. 中国科学院 信息工程研究所，北京 100093；2. 信息内容安全技术国家工程实验室， 北京 100093)

**摘　要**：针对分布式流处理系统中当输入数据的键值呈现倾斜分布时工作节点负载不均衡的问题，提出一种适应元组分布倾斜的流划分策略。该策略使用 lossy counting 算法计算键值出现的频率，以区分热键和非热键，同时，在数据源节点上维护一个多维向量，以记录每个工作节点的负载。当一个键值为热键的流元组到达后，数据源节点查询每个节点的负载，然后将该元组发送给负载最小的节点进行处理。在集成了该流划分策略的分布式流处理系统中运行的实验表明，与按键值划分的策略相比，该策略具有更好的负载均衡性，使系统的吞吐量提升 40%，与随机划分相比，内存开销降低 55%。

**关键词**：倾斜分布；负载均衡；日志流；大数据

# DLSPS:Distributed Log Stream Processing System with Adaptive Stream Grouping

ZHAO Jing-fen[1,2], ZHANG Peng[1,2], SUN Yong[1,2], LIU Qing-yun[1,2]

(1. Institute of Information Engineering CAS, Beijing 100093, China;

2. National Engineering Laboratory for Information Security Technologies , Beijing 100093, China)

**Abstract:** This paper introduces a new stream partitioning scheme to solve the problem of load balancing of stateful applications in distributed stream processing engines when the input stream follows a skewed key distribution. The scheme uses lossy counting algorithm to calculate the frequency of key and identifies the hottest keys in the stream. On the other hand, the source independently maintains a local load-estimate vector with one element per worker to track the load of downstream workers. When a tuple with a hot-key comes, the proposed partitioning scheme will check the load-estimate vector, and then assign this tuple to the worker whose load is smallest, providing a much better load balance. When deployed on a real Storm cluster, comparing to key grouping, the stream partitioning scheme reduces the load imbalance, makes an improvement of up to 40% in throughput, comparing to shuffle grouping, it requires less memory.

**Key words:** skewed distribution; load balance; log stream; big data

## 1　引言

日志来源于用户上网行为、应用系统（如：云服务、网络监控系统、传感器网络、股票行情分析系统等）、操作系统、网络设备（如：路由器、交换机）等，具有体量巨大、异构、价值密度低的特点。日志系统在各种应用系统中均作为基础组件存在，它是学术研究、商业开发的重要数据源[10]。具体体现在：对于系统管理员而言，在系统运行期间，需要根据日志来推断错误发生的原因，用于分析故障根源，发现系统的性能瓶颈，或者用来检测安全问题；对于服务提供商而言，可以通过日志系统记录用户的行为，如：查询、点击、收藏等，基于用户行为数据的分析与挖掘可以用于生成报告、预测市场趋势、发现欺诈行为[8]。

随着互联网的快速发展，尤其是近几年移动互联网的掘起，随之产生的日志数据正在以指数级增长。当日志数据累积到一定的量级，其所蕴含的价值就从量变走向了质变，就可以从中挖掘出很多规律性信息。同时，大数据时代的来临，开源技术生

**基金项目**：国家自然科学基金资助项目（61402464, 61402474, 61602467）

**Foundation Items:** National Natural Science Foundation of China (61402464, 61402474, 61602467)

态圈的逐步完善和广泛应用，使得采用大数据技术构建日志处理系统，进行日志采集、分析、存储，从而最大程度地挖掘其所蕴藏的潜在价值成为可能[4]。

日志的存在形式有两种：日志文件中的日志记录和日志流。日志文件可以通过批量发送文件的方式进行采集，之后利用开源的批处理系统 Hadoop 进行离线分析[4]。日志流是指大量、持续、快速、随时间不断变化的、应用系统产生的用于记录其在使用过程中发生的各种事件的元组序列。日志流具有实时性、无限性、不可再现性、突发性等特点，需要使用流处理系统进行实时分析。

在日志分析过程中，经常需要进行聚集统计，即统计某一属性值与其他取值的映射情况，具体实现方式为：以聚集的属性为键值（key），对数据流进行划分，确保具有相同键值的消息分发至同一个工作节点进行处理，这种流划分策略叫做按键值划分（Key Grouping）[14]。该算法的缺陷是：流式数据的随机性和突发性使数据流中键值呈现倾斜分布，这种划分会导致部分工作节点负载过大，出现严重的不均衡现象，从而影响整个系统的性能。如图 1 所示，元组 tuple 的颜色表示键值，由于 key 的分布不均匀，导致工作节点负载不均衡[2]。流处理系统中广泛使用的另一种分发策略是随机划分（Shuffle Grouping）。它采用轮询机制，能取得理想的负载均衡性，但仅适用于无状态数据，对于有状态数据，需要增加额外的汇聚操作，造成时间和空间的浪费[1]。

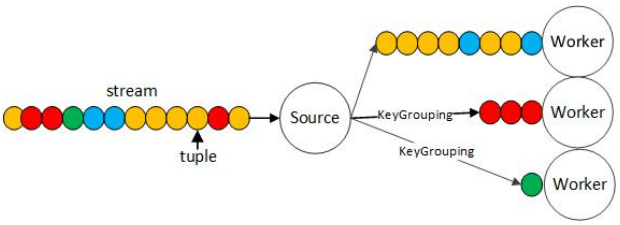


图 1　key 的倾斜分布导致工作节点负载不均衡

本文要解决的关键问题是：在分布式流处理系统中，当输入数据的键值呈现倾斜分布时，工作节点出现负载均衡的问题。为此，本文设计实现了一种新的流划分策略。与 Key Grouping 将一个键值对应的所有消息映射至同一个工作节点不同，该方法依据键值的出现频率将其划分为 hot-key 和 non-hot-key 两类，分配更多的节点处理 hot-key 对应的消息，从而实现负载均分。为了进一步取得更好的负载均衡性，此算法应该将 hot-key 对应的消息发送给若干个候选节点中负载最小的进行处理，前提是已知工作节点的负载，所以本文需要一种高效的负载估计方法。

综上所述，本文的主要工作如下：

- ✧ 提出了一种适应元组分布倾斜的流划分策略，该方法使用 lossy counting 算法区分 hot-key 和 non-hot-key，在 source 节点上维护一个 n 维向量，记录每个工作节点的负载，从而实现将 hot-key 对应的消息发送给负载最小的候选节点处理，进行负载均衡。在 Apache Storm 中运行的实验表明，与 Key Grouping 相比，该算法具有更好的负载均衡性，使系统的吞吐量提升 40%，内存开销仅为 Shuffle Grouping 的 45%。
- ✧ 设计实现了一个分布式日志流处理系统。该系统支持多源日志采集、近实时的在线统计和海量日志的离线统计，具备高可靠、可扩展、高性能、负载均衡等特性。

## 2　系统架构设计

### 2.1　系统架构

分布式日志处理系统基于开源的大数据技术 Flume、Kafka、Storm 等构建，支持多源日志采集、近实时的在线统计和海量日志的离线统计，具备高可靠、可扩展、高性能、负载均衡等特性。如图所示，系统分为日志采集模块、日志传输模块、日志统计模块、存储模块、展示模块。

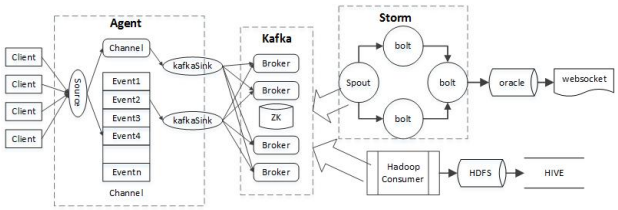


图 2　分布式日志处理系统架构

### 2.2　系统特性

分布式日志处理系统具有如下特性：

可用性：系统采用分布式部署，以集群的形式对外提供服务，部分节点宕机不会影响整个系统的可用性。

可靠性：使用消息队列 Kafka 保证消息的可靠性，并实现了一个可靠的 consumer 的设计方案，确保消息仅被传输一次，不丢失也不重传。

扩展性：采用异步通信的方式降低日志采集模块和统计模块的耦合性，便于扩展；能够方便的增

加和删除日志收集节点以适应产生日志的前端服务器数量的伸缩性变化；日志传输和统计模块可以通过增加节点进行水平扩展，提高整个系统的处理能力。

高性能：高吞吐、低延迟的日志读写、统计能力。

负载均衡：日志流具有突发性和随机性，如果日志流中某个元组呈现倾斜分布，那么分发至并行计算节点的数据量是不均衡的，这将导致部分工作节点负载过重，从而影响整个系统的性能，为此本文提出了一种适应元组分布倾斜的负载均衡策略，这部分将在第三节重点介绍。

### 2.2.1 日志采集模块

本系统使用Cloudera提供的开源分布式海量日志采集系统[15]Flume 作为数据采集模块。该系统可实现多源异构的日志数据收集、传输和聚合，具有分布式、可扩展、容错、高性能的优点。

实际环境中，需要在每一台产生日志数据的前端机上部署 Flume Agent，负责实时采集数据并发送给 Kafka 集群。如果不采用分层架构存在如下问题：若产生日志的前端机数量较多，且分布在不同的网段，分属于不同的业务集群，生成的日志消息要发送给 Kafka 的不同 Topic，前端机上部署的 Flume agent 处于同一层中，不便于分组管理；当 Kafka 集群停机维护或者升级时，需告知所有 Flume agent 所在节点的业务方，做好应对措施，扩展性差。

本系统中数据采集模块使用分层架构，如下图所示：

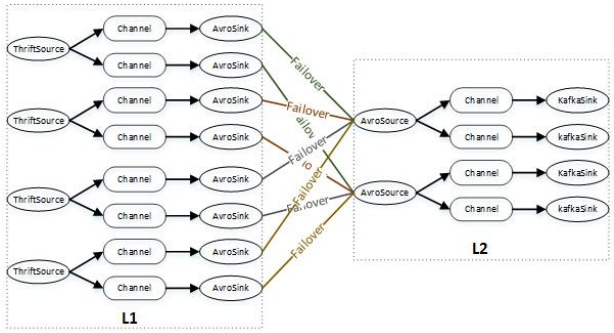


图 3　Flume 分层架构图

第一层（L1 层）是日志采集层，Flume agent 部署在每台前端机上，收集本机生成的日志，发送给第二层。其中，Source 视具体业务而定，channel 使用 Memory Channel，其性能较好，但注意不应该在 channel 中累积过多数据；sink 使用 Avro sink，以连接第二层的 Avro source。

第二层（图中 L2 层）是汇聚层，Flume agent 部署在独立的汇聚节点上（其数量小于前端机），接收来自采集层 agent 的数据，之后发送给 Kafka。汇聚层 agent 选用 Avro source，当 Kafka 集群因故障停机或停机维护升级时，该层应该对日志进行缓冲，且具备较高的稳定性，防止机器故障导致数据丢失，所以采用 File Channel，sink 则是 Kafka sink。

除此之外，为了防止汇聚层机器故障停机时，L1 层发送数据不成功而丢日志，L1 层可以配置 sink group，使用 failover 机制[15]。

### 2.2.2 日志传输模块

日志传输模块位于日志采集模块和日志统计模块之间，它降低了两个模块的耦合度，如果一个模块出现故障，另一个模块可以正常提供服务，保证了系统的高可用。同时，日志传输模块负责缓存日志采集模块预处理后的日志数据，它支持日志数据的异步处理，可达到削减流量的作用。该模块基于分布式发布订阅消息系统 Kafka 实现，前面的日志采集系统 Flume 作为生产者，将日志推入（push）相应的 topic，后端的日志统计模块 Hadoop 和 Storm 作为消费者从 Kafka 中拉取（pull）数据进行消费，实现了日志的离线处理和在线实时统计。

Kafka 是 Linkedin 所支持的一款开源的、分布式、高吞吐量的发布订阅消息系统[13]，将生产者(producer)、代理服务器(broker)和消费者(consumer)部署在多台机器上，构成显式的分布式系统。

作为数据业务的基础支撑系统，除了应该满足高并发和高可用以外，数据的质量即数据可靠性也至关重要。但是由 Kafka 原生提供的 consumer API 将消费数据的结果与消费状态 offset 分开独立存储，并不能够保证数据的可靠性（不支持 exactly once）。所以本文提出了一个可靠的 consumer 的设计方案，体现为系统中的 HadoopConsumer 功能模块，该方案基于 Kafka 的 low-level API 实现，将消费结果和 offset 均存储至 HDFS 相应文件内，保障了消息的可靠性。

Kafka 提供的消息传递可靠性保证有三种：At most once：消息可能丢失但绝不重传；At least once：消息不会丢失，可能重传；exactly once：最理想的状态，消息只被传送一次，不丢失也不重传，Kafka 目前不能保证。Kafka 中由 consumer 自己维护消费

状态，定期提交 offset 至 ZooKeeper 或者特定 topic（依据 Kafka 版本不同而不同），consumer 拉取一条消息、进行处理、更新 offset 值可以有多种组合方式，导致不同的可靠性：

① consumer 拉取消息，更新 offset，最后消费消息保存结果。可能消息未处理之前 consumer 因故障宕机，但 offset 已被更新，consumer 重启后或者新的 consumer 会从 offset 之后的位置继续消费，造成数据丢失。

② consumer 拉取消息，消费消息保存结果，最后更新 offset。存在的可能情况是：consumer 在保存消费结果后更新 offset 之前宕机，会导致消息的重复消费。

③ 将 offset 和消费消息后的输出同步存储，例如都保存在 HDFS 文件中，就可以保证他们被同时更新，间接实现 exactly once 级别的可靠性。

具体方案如下：

HadoopConsumer 功能模块基于 Kafka low-level API 和 HDFS API 开发实现，通过读取配置文件获得要消费的 topic 列表和 Kafka broker 的 IP 地址，之后针对每个 partition 创建一个线程作为 consumer 消费消息，并将消息追加至 HDFS 的相应文件内，同时更新 offset 值。该模块由两个子模块构成，其中 DataGenerator 仅在第一次运行 HadoopConsumer 时调用，完成一系列的初始化工作，SimpleKafkaETLJob 负责消费消息，追加 HDFS 文件，更新 offset.dat 文件。

DataGenerator 为配置文件中每个 topic 的每一个 partition 创建如下相关的文件。

①文件 offset.dat，存储 topic、partition、leader broker IP、Port、以及 offset 值；offset 初始值为-1，SimpleKafkaETLJob 运行完毕后会更新该文件，修改 offset，以便之后继续消费；

②HDFS 不支持逐条记录的流式追加，本方案中将消费到的数据先写入本地文件，之后将整个文件追加至 HDFS，所以 DataGenerator 还需创建进行追加的文件 message.txt；

HadoopConsumer 模块输出结果展示：表明从 topic：bigdata 第 0 个分区的 2784 条消息开始消费，leader broker 的 IP 地址为 192.168.11.179；本次消费的 offset 范围是 2783-2783，共消费到 0 条数据，也就是生产速率落后于消费速率；将消费到的消息写入本地文件 message.txt，然后此文件批量追加至 HDFS 中，同时更新 offset.dat，便于下一次继续消费；

```
<terminated> KafkatoHDFSMain (2) [Java Application] /opt/jdk/jdk1.8.0_60/bin/java (2015-11-18 上午1
KafkatoHDFSMain be called!
threadCount1
Setting hadoop jar file for class:class kafka.etl.impl.SimpleKafkaETLJob  to null
*****************************************************************************
          Running on Real Hadoop Cluster(local)
*****************************************************************************
Init request from tcp://192.168.11.179:9797     bigdata 0       2783
Update starting offset with 2783
Using offset range [2783, 2783]
the_last_offset=2783
Totally consume messages _readBytes = 0
Write messages into local file Success!
Append local file to hdfs Success!
Updata offset.dat Success!
```

图 4 HadoopConsumer 模块输出结果

### 2.2.3 日志统计模块

该模块的功能是作为消费者消费 Kafka 消息队列中的数据，过滤、处理每条日志信息，最后把统计结果写入存储模块。日志统计模块包括实时流处理和离线批处理两部分。实时流处理基于 Storm 设计实现，离线批处理使用 Hadoop 分布式文件系统 HDFS 和 Hive 数据仓库实现。

Storm 是 Twitter 支持开发的一款开源、分布式、实时的大数据流式计算系统[14]。任务拓扑(topology)是 Storm 的逻辑单元，一个实时应用的计算任务将被打包为任务拓扑后发布，任务拓扑是由一系列 Spout 和 Bolt 构成的有向无环图，其中 Spout 负责从外部数据源不间断地读取数据，本系统中从 Kafka 处拉取日志，并以 Tuple 元组的形式发送给相应的 Bolt；Bolt 负责对接收到的数据流进行计算，实现过滤、聚合、查询等具体功能，可以级联，也可以向外发送数据流[9]。数据流由 Tuple 元组序列构成，它按照指定的流划分策略在拓扑中流动。

Storm 系统的不足之处是：任务拓扑的结构无法依据输入动态调整[11]，无法适应数据负载的动态变化，如果数据流中某个元组呈现倾斜分布，那么分发至并行计算节点的数据量是不均衡的，这将导致部分工作节点负载过重，从而影响整个系统的性能。为此，下文将介绍一种适应元组分布倾斜的流划分策略。

离线处理方案：除了实时统计之外，实际项目中也会有大规模离线统计的需求，这些应用场景对时效性要求不严格，但是需要对大量的历史数据进行汇总统计，产出数据统计报表。本系统中，基于 Kafka low-level API 和 HDFS API 开发实现了一个 HadoopConsumer 功能模块，该模块作为消费者从 Kafka 中拉取数据，之后将数据存储在 HDFS 文件内，编写 HQL 语句进行查询。

## 3 适应元组分布倾斜的流划分策略

### 3.1 数据流模型

运行在 Storm 集群中的任务拓扑是由 Spout 和 Bolt 构成的有向无环图[6]，其中顶点代表工作节点，负责处理流元组，边代表数据通道，上游节点的输出流经通道成为下游节点的输入，元组序列按照指定的流划分策略在拓扑中流动。

简单起见，本文只分析数据流的一次流动，上游节点集用 $S$ 表示，下游节点集用 $W$ 表示，节点个数分别为 $|S|$、$|W|$。

每一条消息表示为 $\mathrm{m}=\langle t,k,v\rangle$，$t$ 为消息到达的时间戳，消息的键值 $k\in K$，$|K|=K$ 是所有消息的 key 组成的集合，$v$ 是消息内容。

流划分策略（也叫数据分区方法）：$P_{\mathrm{t}}:K\rightarrow N$，将 key 映射至一个整型值，也就是处理此类消息的工作节点。

工作节点 i 在时刻 $t$ 的负载：

$$\mathrm{L_i}(t)=\left|\{\langle\tau,k,v\rangle:P_\tau(k)=i\wedge\tau<t\}\right|$$

本文定义时刻 $t$ 系统的不均衡值[3]为：

$$\mathrm{I}(t)=\max_i(L_i(t))-\underset{i}{avg}(L_i(t)),i\in W$$

### 3.2 流划分策略

被广泛应用的 Key Grouping 仅当数据流中键值呈现倾斜分布时，才会导致分发给每个工作节点的负载不均衡，影响系统性能。这是因为 Key Grouping 使用一个哈希函数对所有数据流进行划分，未考虑键值的出现频率，导致频率较高的 key 对应的工作节点负载过重。所以本文提出的算法将 key 按照出现频率划分为 hot-key、non-hot-key 两类，hot-key 可使用多个哈希函数，将其消息分发至更多的工作节点进行处理，达到分流的目的。该算法需要解决如下问题：

① 该算法应用于有状态数据的流划分，将一个键值对应的消息分发至多个工作节点处理，为了得到正确的结果，不得不增加汇聚操作，导致额外的时间和空间开销，所以需要评估算法的内存使用情况；

② 需要定义一个合理的频率阈值区分 hot-key 和 non-hot-key；

③ hot-key 对应的消息分发给 X 个工作节点，X 是一个固定的值或者随键值的频率不同而变化；

④ 当一个键值为 hot-key 的流元组到达后，source 应该将其划分至 X 个候选工作节点中负载最小的节点进行处理，所以 source 需要知道工作节点的负载信息。

为了实时统计流元组的频率，本文在 source 节点上运行 Lossy Counting 算法[5]。Lossy Counting 是由哈佛大学的 Manku 和 Motwani 提出的第一个近似计算流数据频繁度的算法。此算法简单高效，只需对样本集合进行一次扫描，时间复杂度为 $O(\log\varepsilon N)$，空间复杂度为 $O(1/\varepsilon\log\varepsilon N)$ [12]。用户指定两个输入参数：一个支持度阈值 $s\in(0,1)$，另一个误差范围 $\varepsilon\in(0,1)$。输出结果满足：所有真实频率超过 sN 的数据项均被输出、所有真实频率低于 $(s-\varepsilon)N$ 的数据项均不输出、算法估计的频率值与真实频率的误差小于 $\varepsilon$N。N 是当前数据流中的元组总数。

综上可知，设置合理的支持度阈值 s，就可以利用 lossy counting 算法划分出 hot-key 和 non-hot-key。由于本研究实现了一个分布式日志流处理系统，数据源是在线捕获国内某教育网关和企业网关的用户上网流量，进行协议解析之后生成的日志流。所以综合考虑需要在 Storm 中运行的业务逻辑以及数据源中相关字段的分布情况，本文设置支持度阈值 $s=0.05$，也就是出现频率大于 N/20 的键值为 hot-key，否则为 non-hot-key。并规定，对于 non-hot-key，使用两个不同的哈希函数为其选择两个候选工作节点；对于 hot-key，使用 $|W|/5$ 个不同的哈希函数为其选择 $|W|/5$ 个候选工作节点，如果流处理系统中 worker 较少导致 $|W|/5\le 2$，则 hot-key 也随机映射至两个工作节点。

除此之外，本文提出一个简单有效的工作节点负载估计方法：在每个 source 节点上维护一个 $|W|$ 维数组，记录该 source 加载给每个工作节点的负载，也就是消息条数，从而巧妙地避免了向 worker 询问负载信息，节省通信开销。所以，当一个键值为

hot-key 的流元组到达后，source 首先根据哈希函数计算出 $|W|/5$ 个候选节点，之后查询负载数组，将该元组发送给负载最小的节点进行处理。

### 3.3 实验分析与验证

#### 3.3.1 数据集

数据集是在线捕获国内某教育网关（EDU）和某企业网关（ISP）24 小时内的用户上网流量，进行协议解析之后生成的日志流。日志流包括用户访问的网页 URL、域名、IP 地址等信息，其中域名作为键值。

数据集内域名的分布情况概述：

EDU 中约有 75%的域名访问量小于 1000，仅有 4.39%的域名访问量超过 10000，Top10 域名的访问总和占数据集总量的 12.5%；

ISP 中访问量小于 1000 的域名占 74%，访问量大于 10000 的占 3.79%，Top10 域名的访问总量占 7.7%。

可见，数据集内键值呈现倾斜分布。

表 1　数据集介绍

| 数据集 | 位置 | 日期 | 规模 |
|---|---|---|---|
| EDU | 北京某教育网关 | 2016/08/21 | 日志总量：32M<br>不同键值个数：12,078<br>出现频率最高的 key 的频率值：7.42% |
| ISP | 北京某企业网关 | 2016/08/21 | 日志总量：25M<br>不同键值个数：15,104<br>出现频率最高的 key 的频率值：9.01% |

#### 3.3.2 实验结果

为了验证本文提出的适应元组分布倾斜的流划分策略（下文简称 KG2：改进的 Key Grouping 算法），本文设计了一个简单的 topology，基于 EDU 和 ISP 两个数据集进行实验，回答如下问题：

1) 采用 KeyGrouping、KG2、ShuffleGrouping 三种流划分策略，分发至工作节点的负载以及内存开销有何差异；
2) KG2 算法能否提高系统的吞吐量；

如下图所示，KafkaSpout 负责从 Kafka 中提取数据，分发给 CounterBolt 进行实时统计，下游的 AggregatorBolt 汇总每个域名的访问次数。

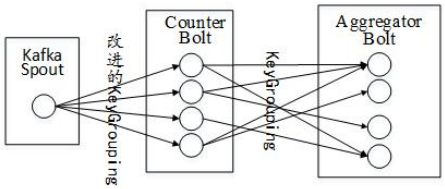


图 5　Storm 计算拓扑结构

实验运行于 9 台物理机（一个 master，8 个 slave）组成的 Storm 集群上。服务器的配置为：8vCPU，Intel(R) Xeon(R) CPU E5-2680 v2@ 2.80GHz，内存 16 GB，磁盘 500 GB。Supervisor 的并行度设置为 8，所以集群中共有 8*8=64 个 worker。

**A：负载均衡性**

表 2 显示了系统的负载不均衡值随工作节点个数的变化情况。Key Grouping 的均衡性最差，Shuffle Grouping 取得最好的负载均衡性，但它仅适用于无状态数据，对于有状态数据，需要增加额外的汇聚操作，造成时间和空间的浪费，所以通常将其作为评估其他流划分策略性能优劣的基准；本文提出的 KG2 算法性能仅次于 Shuffle Grouping。同时可以看出，随着集群中工作节点个数的增加，不均衡值也逐渐增大，因为更多的工作节点导致分发给每个节点的键值个数减少，所以更难均衡 hot-key 和 non-hot-key 导致的负载差值。

表 2　系统的不均衡值随工作节点个数的变化情况

| 数据集 | EDU | | | | ISP | | | |
|---|---|---|---|---|---|---|---|---|
| W | 5 | 10 | 20 | 50 | 5 | 10 | 20 | 50 |
| ShuffleGrouping | 0.70 | 0.93 | 1.28 | 1.89 | 0.45 | 0.71 | 1.25 | 2.07 |
| KeyGrouping | 1.52 | 1.74 | 2.06 | 2.17 | 3.28 | 3.20 | 3.79 | 4.10 |
| KG2 | 0.80 | 1.22 | 1.44 | 2.11 | 0.52 | 0.53 | 1.34 | 2.07 |

**B：内存开销**

内存开销分为两部分：source 的内存开销和 worker 的内存开销。

Source 负责从 Kafka 中提取数据，分发给 worker，他维护着一个 $|W|$ 维的负载向量（$|W|$ 为工作节点的个数），记录该 source 加载给每个工作节点的负载，所以 source 的内存开销为 $O(|W|)$。

worker 的内存开销：对于有状态数据，为了得到正确的统计结果，本文提出的流划分策略和 Shuffle Grouping 都需要增加汇聚操作，导致额外的内存开销，所以主要对比这两种算法。Shuffle

Grouping 在最坏情况下，一个 key 对应的消息分发给集群中每个 worker，内存开销为 $|K|\times|W|$。KG2 中，hot-key 的消息分发至 $|W|/5$ 个 worker，non-hot-key 分发给 2 个 worker，所以内存开销为 $|hot\text{-}key|\times|W|/5+|non\text{-}hot\text{-}key|\times 2$。

图 6 展示了 KG2 相对于 Shuffle Grouping 的内存开销随时间的变化情况。集群中的 worker 个数为 8*4=32。可见，KG2 占用的内存远远小于 Shuffle Grouping，约为其 45%。

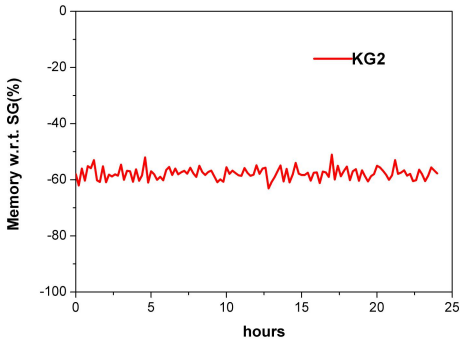


图 6　KG2 相对于 SG 的内存开销随时间的变化情况

**C：系统吞吐量**

实验中，采用三种不同的流划分策略产生的系统吞吐量如下图所示。KG 对应的吞吐量是最低的，因为他会引起工作节点负载的严重不均。改进版 KG2 的吞吐量接近于 SG，比 KG 提升了 40%，充分表明本算法的有效性。

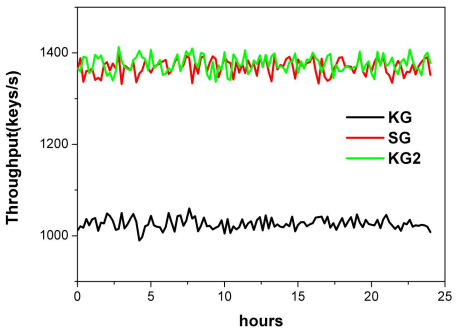


图 7　Storm 系统的吞吐量

## 4　结论

本文介绍了一个适应元组分布倾斜的分布式日志流处理系统。本系统主要由日志采集模块、日志传输模块、日志统计模块组成。同时针对当输入数据的键值呈现倾斜分布时，工作节点出现负载均衡的问题，提出一种流划分策略。该方法利用 lossy counting 算法区分 hot-key 和 non-hot-key，在 source 节点上维护一个 n 维向量，记录每个工作节点的负载，从而将 hot-key 对应的消息发送给负载最小的候选节点进行处理，实现负载均衡。在 Apache Storm 中运行的实验表明，与 Key Grouping 相比，该划分策略具有更好的负载均衡性，使系统的吞吐量提升 40%。

本研究的下一步工作是继续改进该流划分策略，当输入的数据流极端倾斜，或者该策略部署运行于一个大规模的分布式集群时，都能取得理想的负载均衡性。

**参考文献：**

[1] B. Gedik, Partitioning functions for stateful data parallelism in stream processing,The VLDB Journal, pp. 1－23, 2013.

[2] Anis Uddin Nasir M, De Francisci Morales G, Garcia-Soriano D, et al. The power of both choices: Practical load balancing for distributed stream processing engines[C]//Data Engineering (ICDE), 2015 IEEE 31st International Conference on. IEEE, 2015: 137-148.

[3] Anis Uddin Nasir M, De Francisci Morales G, Kourtellis N, et al. When Two Choices Are not Enough: Balancing at Scale in Distributed Stream Processing[J]. arXiv preprint arXiv:1510.05714, 2015

[4] Logothetis D,Trezzo C,Webb KC, et al. In-situ MapReduce for Log Processing[C] //Proceedings of USENIX Annual Technical Conference. Berkeley, CA,USA: USENIX Association, 2011:115－129.

[5] Gurmeet Singh Manku , Rajeev Motwani, Approximate frequency counts over data streams, Proceedings of the 28th international conference on Very Large Data Bases, p.346-357, August 20-23, 2002, Hong Kong, China

[6] M.Cherniack, H.Balakrishnan,M.Balazinska,D.Carney,U. Cetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In CIDR, volume 3, pp. 257－268, 2003.

[7] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams.In ICDT, pp. 398－412,

2005.

[8] 屈国庆. 基于 Storm 的实时日志分析系统的设计与实现[D].南京大学,2016.

[9] 程学旗,靳小龙,王元卓,郭嘉丰,张铁赢,李国杰. 大数据系统和分析技术综述[J]. 软件学报,2014,09:1889-1908.

[10] 周心林. 数据流上多聚集查询的优化技术[D].苏州大学,2012.

[11] 孙大为,张广艳,郑纬民. 大数据流式计算:关键技术及系统实例[J]. 软件学报,2014,04:839-862.

[12] 王伟平,李建中,张冬冬,郭龙江. 一种有效的挖掘数据流近似频繁项算法[J]. 软件学报,2007,04:884-892.

[13] Kafka,distributed publish-subscribe messaging system. 2016. http://data.linkedin.com/opensource/Kafka/

[14] Storm wiki. 2016. http://en.wikipedia.org/wiki/Storm

[15] Flume 2016, http://flume.apache.org/

**作者简介：**

赵静芬，硕士研究生，中国科学院信息工程研究所，北京市海淀区闵庄路甲 89 号，100093，13261595496，zhaojingfen@iie.ac.cn

张鹏，副研究员，中国科学院信息工程研究所，北京市海淀区闵庄路甲 89 号，100190，15801683810，pengzhang@iie.ac.cn