流数据实时接收方案的研究

张笑燕，刘志浩，杜晓峰，陆天波
（北京邮电大学计算机学院（国家示范性软件学院），北京 100876）

摘 要：针对现代数据仓库系统中常见的需接收大量流数据，且其与磁盘上已有的数据做连接后再入库的场景进行了探讨。通过合理设置磁盘分页和应用缓存模块，分散磁盘 I/O 压力，在已有研究的基础上提出了一种具有更高效率的数据接收方案，并引入一致性哈希函数将其扩展到分布式环境，提出一种应用于分布式环境的 D-CACHEJOIN 算法。通过理论计算算法的成本模型，并使用服从 Zipfian 分布的数据进行模拟实验。实验结果表明，在接近现实的实际应用场景下，所提算法拥有比现有算法更高的效率，同时能够快速方便地扩展到分布式环境。

关键词：流数据；缓存；分布式系统；一致性哈希函数
中图分类号：TP3-0
文献标志码：A
DOI: 10.11959/j.issn.1000−436x.2022080

Research on a real-time receiving scheme of streaming data

ZHANG Xiaoyan, LIU Zhihao, DU Xiaofeng, LU Tianbo
School of Computer Science (National Pilot Software Engineering School), Beijing University of Posts and Telecommunications, Beijing 100876, China

Abstract: Discussing the common scenarios in modern data warehouse systems that need to receive a large amount of streaming data, connect it with the existing data on the disk, and then store it in the warehouse. By rationally setting disk paging and applying cache modules to disperse the disk I/O pressure, a more efficient data receiving scheme was proposed based on the existing research, and a consistent Hash function was introduced and extended to distributed environment and a D-CACHEJOIN algorithm applied to distributed environment was proposed. The cost model of the algorithm was calculated by theory and simulation experiment was performed using data that obey the Zipfian distribution. The experiment results show that the proposed algorithm has higher efficiency than existing algorithms in practical application scenarios close to reality, and can be quickly and easily extended to distributed environments.

Keywords: streaming data, cache, distributed system, consistent Hash function

0 引言

随着信息社会飞速发展，数据产生的速度越来越快。现实中存在一类常见的业务，即将大量源源不断到达的流数据 S 先与存储在磁盘上的关系表 R 进行连接，对 S 进行半流连接关联更新操作（如去重、修正等）后再将其写入数据仓库[1]。该过程被定义为 $S\infty_C R$[2]。其中，C 代表某种半流连接操作。

由于源数据系统的不同，逻辑上相同的元组可能具有不同的值，在数据进入数据仓库时，需要通过关联表进行统一，保证数据的一致性。

为了更具体地理解这种半流连接关联更新操作，图 1 展示了数据仓库中涉及该操作的一个例子。由于数据来源的不同或数据的延迟到达，系统中同一个 id 可能具有不同的名称，在数据进入数据仓库之前，需要使用一张关系表将 id 转换为系统内部的 id，对不同的名称进行统一。

在真实的数据仓库系统中，R 保存在磁盘上，一般占用空间较大，无法全部放入内存；S 包含不断到达的流数据，其中的每个元组都需要和 R 进

收稿日期：2022−01−05；**修回日期**：2022−04−05
基金项目：国家自然科学基金资助项目（No.62162060）
Foundation Item: The National Natural Science Foundation of China (No.62162060)

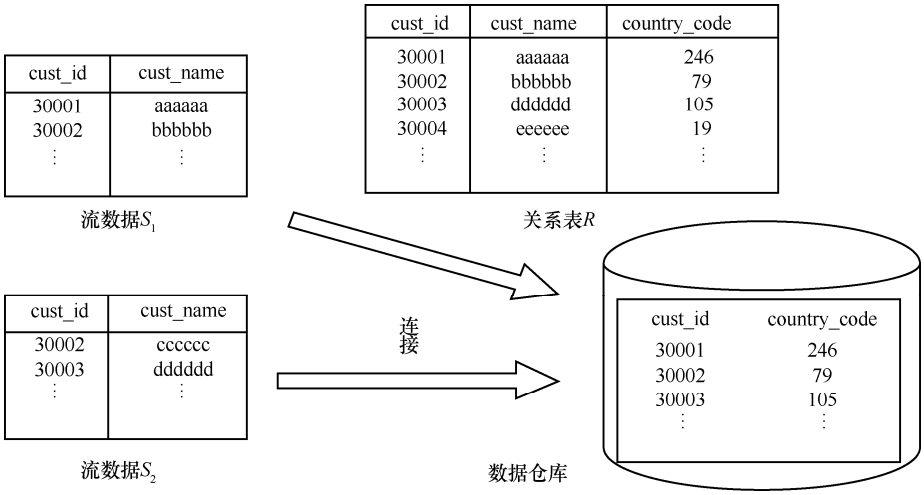


图 1　数据仓库关联更新示意

行连接操作[3-5]。这便产生了一个问题：S 中的元组以流数据的形式不断快速到达，而 R 中的元组需要通过磁盘 I/O 以相对较低的速率读取，造成更新时需要消耗大量时间等待磁盘 I/O，导致数据处理的延迟。

事实上，为了解决流数据连接操作中的各种问题，与此有关的研究从未中断。Polyzotis 等[6]提出了 MESHJOIN 算法，通过对关系表的分组读取和对流数据的分页读取，把对磁盘的 I/O 时间分摊到了若干个流数据元组的读取中，平衡了关系表的读取速率和流数据的到达速率的差异。但是 MESHJOIN 算法并没有考虑流数据 S 的特征和关系表 R 的组织，因此在处理倾斜数据时表现较差[7]。Vaidehi 等[8]提出了分布式嵌套循环连接处理（DNLJP, distributed nested loop join processing）算法，对流数据的来源进行分组，根据其所定义的不同查询成本、查询时间等对连接查询操作进行优化，降低了分布式集群中各节点之间通信的成本，提高了对海量数据连接查询的效率。DNLJP 是可以用于分布式集群的算法，而 MESHJOIN 和 CACHEJOIN 算法只适于单机环境，无法直接应用于分布式环境[9-10]。Naeem 等[11]提出的 CACHEJOIN 算法考虑了数据倾斜的问题，通过引入一个缓存，将关系表中较常用的部分存储在缓存中，其中的元组有较高的概率和流数据匹配成功，减少了磁盘的 I/O 次数。Jeon 等[12]提出了 DS-join 方案，通过使用几种流处理引擎的微批处理模式，控制数据分区，减少了各节点之间的网络通信频率，同时引入缓存模块对连接操作并行处理，减少磁盘 I/O 次数，并且拥有自动调整缓存区大小的优化设置。为了保证分布式流连接操作中处理结果具有完整性和一致性，Yuan 等[13]测试了数据仓库接收到的流数据出现错误时连接操作结果的质量优劣，提出了基于有序传播模型的、应用于分布式流连接系统的 Eunomia 方法，保证了所有连接器的元组到达顺序的一致性，能够消除数据传播过程中的某些错误，提高了分布式流连接处理的扩展性和吞吐率。狄程等[14]设计并实现了对多源异构流数据的处理系统，在一定程度上消除了数据的结构不同对数据引入、连接处理造成的复杂问题，使流处理服务化、模块化，减少了流数据处理程序的开发成本。

因此，本文在已有研究的基础上进行改进和优化，提出了应用于分布式环境的 D-CACHEJOIN 算法。D-CACHEJOIN 算法的关键在于采用一致性哈希函数[15]的策略，将 R 分区存储在不同节点，并在后续的匹配过程中使用这一计算结果，达到快速将 R 与 S 进行匹配的目的。同时，本文使用服从 Zipfian 分布的模拟数据[16-17]，定量计算算法执行的成本开销，以优化算法的执行效率[18]。实验表明，在拥有一定的可用内存时，D-CACHEJOIN 算法在处理流数据时具有良好的实时处理性能，同时具有易于扩展的特性。

1　基于缓存的分布式 D-CACHEJOIN 算法

1.1　D-CACHEJOIN 算法的原理描述

与 CACHEJOIN 算法类似，D-CACHEJOIN 算法拥有 2 个阶段——流检测阶段和磁盘检测阶段。流检测阶段使用流数据 S 作为输入，磁盘检测阶段

使用关系表 R 作为输入，由于内存大小的限制，两阶段每次都分别只处理 S 和 R 的一部分。

D-CACHEJOIN 算法执行架构如图 2 所示[11]，关系表 R 和流数据 S 是外部输入，H_S 用于分次存储 S 中的元组，实际中占用较大的空间；H_R 是缓存模块，用于存储 R 中匹配频繁的元组[19]。

下面对本文算法的执行架构进行说明。算法在流检测阶段，读取流缓冲区的数据元组并与 H_R 中存储的高频访问元组进行连接匹配，如果在 H_R 中找到所需的元组，则算法生成该元组作为输出，只有在无法匹配时，才会将该元组存储到 H_S 中并将其指针加入队列，以便进行后续的匹配操作。当 H_S 满或 S 中已无数据元组时，流检测阶段结束，磁盘检测阶段开始。在磁盘检测阶段，算法将 R 中的部分元组读取到内存，得以分摊消耗资源较多的磁盘 I/O 操作。读取完成后，算法会将其与 H_S 中的元组进行匹配，匹配成功后，算法生成连接后的元组作为输出。与此同时，匹配次数超过阈值的元组会被缓存到 H_R 中用于流检测。经历数次迭代后，H_S 中最先进入的元组已经完成了和 R 中所有元组的匹配，算法会将其删除，此时 H_S 中有了新的空闲位置，可以继续存储 S 中待匹配的元组，因此算法接下来再次切换到流检测阶段。

D-CACHEJOIN 算法的伪代码实现如算法 1 所示[11]，其中 w 为流检测阶段读取 S 的元组数，b 为磁盘检测阶段读取 R 的元组数。

算法 1 D-CACHEJOIN

输入 关系表 R 和流数据 S

输出 $S \infty R$

```
1)  while(true) do
2)    在流缓冲区读取 w 个元组
3)    for  w 中每个元组 t  do
4)       if  t ∈ H_R  then
5)          输出  t
6)       else
7)          把 t 加入 H_S 并将指针加入队列
8)       end if
9)    end for
10)   在磁盘缓冲区读取 b 个元组
11)   for  b 中每个元组 r  do
12)      if  r ∈ H_S  then
13)         输出 r
14)         f := r 与 H_S 匹配的次数
15)         if( f ≥ thresholdValue ) then
16)            将 r 置换入 H_R 中
17)         end if
18)      end if
19)   end for
```

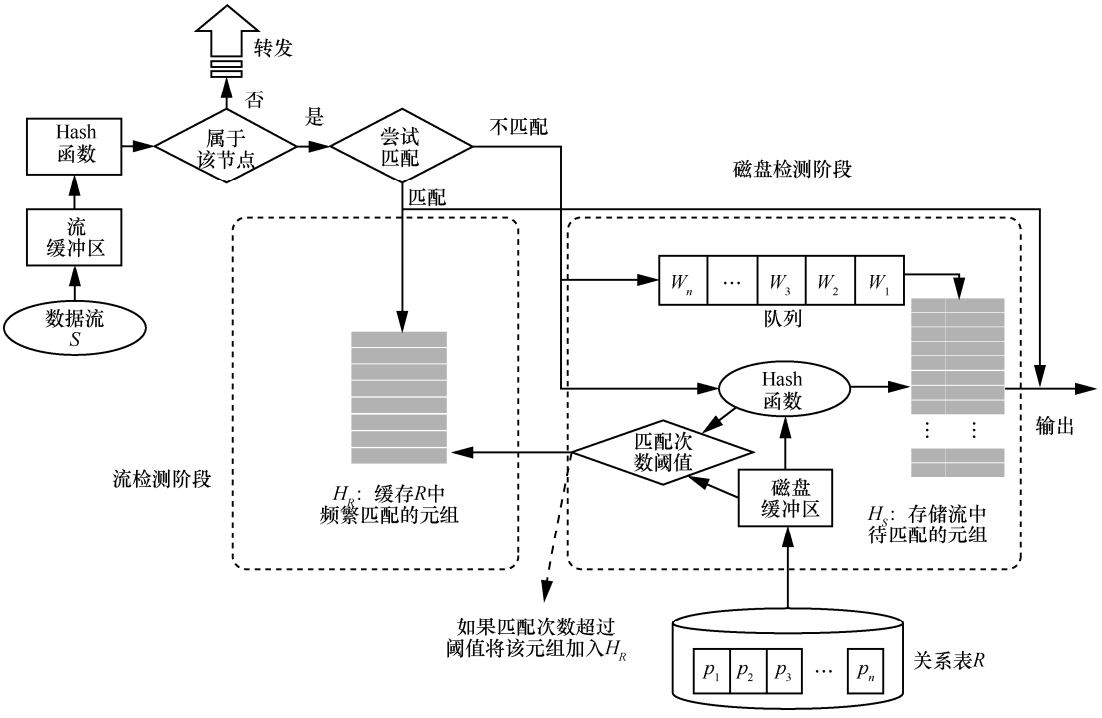


图 2 D-CACHEJOIN 算法执行架构

20) 删除 H_S 中匹配完成的元组及其队列中对应的指针

21) end while

为了完整地处理流数据，算法的外层是无限循环，主要包括 2 个部分，流检测阶段和磁盘检测阶段，二者交替进行。

在步骤 2)~步骤 9)的流检测阶段，算法在流缓冲区读取 w 个元组，对 w 中的每个元组 t，算法首先检测检测 t 是否在 H_R 中，如果是则将其连接并输出结果；否则将 t 加入 H_S 中，同时在队列中加入 t 的指针。

在步骤 10)~步骤 20)的流检测阶段，算法在磁盘缓冲区读取 b 个元组，对 b 中的每个元组 r，算法首先检测 r 是否在 H_S 中，如果是则将其连接并输出结果；否则将忽略该元组。由于 H_S 需要存储流中的元组，因此 H_S 是一个多映射，r 可能会成功匹配到 H_S 中的多个元组，记该成功匹配的次数为 f；如果某个 r 对应的 f 大于阈值 thresholdValue，就将 r 加入 H_R 中，如果 H_R 已满，则先随机删除 H_R 中的一个元组，然后再将 r 加入 H_R。在每一次的迭代中，如果缓存 H_R 未满，说明阈值设置过高，此时可以自动降低阈值；如果过于频繁地从 H_R 中删除元组，说明阈值设置过低，此时可以自动提高阈值。

1.2 D-CACHEJOIN 算法的系统实现

1.1 节说明了本文算法在单节点环境中的一般过程，本节将算法应用到分布式环境中。为了叙述简洁，本节以拥有 2 个节点的数据仓库为例。

数据仓库接收从数据源经网络传输的流数据时，数据会随机地分配到各节点上。数据向各节点的随机分配是为了实现集群整体上各节点抽取–转换–加载（ETL, extract-transform-load）的负载均衡，在实际处理中，可以在随机选取第一个节点后，按照某个固定的顺序依次选取各个节点作为数据的接收方，消除因选取节点而产生的额外的 CPU 计算开销。节点在接收到数据元组后，直接通过一致性哈希函数计算，如果元组属于该节点，就将其保留，否则将元组转发到对应的节点，同时将计算得到的哈希值一并发送，避免后续产生重复的计算。D-CACHEJOIN 算法的整体架构流程如图 3 所示。对于 MESHJOIN 算法和 CACHEJOIN 算法，由于其相当于只在图 3 中的“连接”环节发挥作用，而数据元组在分布式环境中的接收、转发等都在此之前就已经完成，因此可以相对简单地一并扩展到分布式环境之中。

在基于本文算法的处理流程中，流数据 S、关系表 R 和数据仓库集群中各节点均采用同一个哈希函数进行映射，有利于集群的扩展。同时对于节点接收到的不属于自身、需要转发的元组，通过计算得到的哈希值将会随元组一同转发，避免了重复的计算。

从图 3 可以看出，数据仓库将流数据随机地依次分配给集群中的任一节点，从宏观上看，集群中的所有节点在某一时间段内同时读取数据并进行计算，保留属于自身的数据、转发属于其他节点的数据。这样的设计没有属于中心地位的节点，发挥了分布式集群并行处理数据的优势。数据不需要首先经过中心节点的处理，而是直接在各节点之间进行流转，避免了由于存在中心节点导致系统出现性能瓶颈的问题。

2 D-CACHEJOIN 算法实验分析

2.1 性能评估模型

对于数据仓库的 ETL 过程而言，主要关注的性

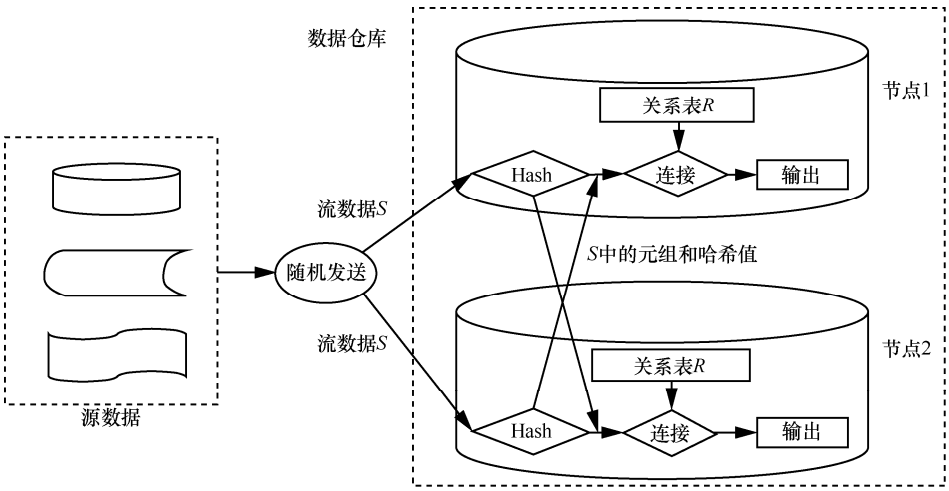


图 3　D-CACHEJOIN 算法的整体架构流程

能指标包括系统吞吐率、内存消耗等。本文研究中使用的分布式系统中，各个节点的作用主要是参与计算和数据存储等，大量运算如备份、保证数据一致性等只在内部进行，对内提供支持，对外只提供将多节点集群环境虚拟为单节点环境的接口，集群可以视为一个单节点环境。引入分布式系统直接导致算法增加的开销是一致性哈希函数计算的开销，为了实现负载均衡，令各节点随机接收流数据，然后相互转发数据。文献[6]中已经给出 MESHJOIN 算法的成本模型，本文参考该模型，建立 D-CACHEJOIN 算法的性能评估模型，模型参数如表 1 所示。

表 1　　算法性能评估模型参数

参数	含义
n_{SP}	流检测阶段每次迭代处理元组数量
n_{DP}	磁盘检测阶段每次迭代处理元组数量
v_P /byte	关系表 R 中每个分页的大小
v_R /byte	关系表 R 中元组的大小
a	磁盘缓冲区分页数
n_{RB}	磁盘缓冲区中元组个数，$n_{\mathrm{RB}}=a\frac{v_P}{v_R}$
b	H_R 的分页数
n_{h_R}	H_R 中元组个数，$n_{h_R}=b\frac{v_P}{v_R}$
n_R	关系表 R 中元组个数
λ	流数据 S 中元组在滑动窗口中的个数
σ	关系表 R 与 H_S 中元组的连接成功率
$C_{\mathrm{I/O}}(k)$	读取关系表 R 中 k 个分页的开销
C_E	匹配完成后的一个元组及其指针移除的开销
C_S	读取一个元组进入流缓冲区的开销
C_A	将一个元组加入 H_S 的开销
C_H	匹配 $\frac{wN_g}{b}$ 个元组的开销
C_O	结果输出的开销
C_C	进行一致性哈希函数计算的开销
C_{Ti}	流数据从源端传递到随机节点的开销
C_{Te}	分布式系统中节点间传输的开销

根据表 1 中的参数，下面逐个计算每个过程的开销，最终得到 D-CACHEJOIN 算法的整体吞吐率 μ（单位为元组/秒）。

1) 数据在网络中传输的开销。假设分布式系统由 n 个节点构成，单位时间内需转发的节点数为 $\lambda\frac{n-1}{n}$，因此网络中数据总的传输开销为 $\lambda\left(C_{\mathrm{Ti}}+\frac{n-1}{n}C_{\mathrm{Te}}\right)$。

2) 一致性哈希函数计算的开销为 $\frac{1}{n}C_C$。

3) 读取关系表 R 中 k 个分页的开销为 $C_{\mathrm{I/O}}(k)$。

4) 移除 w 个匹配完成的元组及其指针的开销为 wC_E。

5) 从流缓冲区中读取 w 个元组的开销为 wC_S。

6) 将 w 个元组加入 H_S 的开销为 wC_A。

7) 将 H_S 中元组与 R 中元组匹配的开销为 $k\frac{v_P}{v_R}C_H$。

8) 输出结果的开销。结果输出时的开销和连接的成功率有关，成功率越高，需要输出的结果元组就越多，因此，最终输出结果的开销为 $\sigma k\frac{v_P}{v_R}C_O$。

9) 一次处理过程的总时间消耗为 $T_{\mathrm{cost}}=\lambda\left(C_{\mathrm{Ti}}+\frac{n-1}{n}C_{\mathrm{Te}}\right)+\frac{1}{n}C_C+C_{\mathrm{I/O}}(k)+w\left(C_E+C_S+C_A\right)+k\frac{v_P}{v_R}C_H+\sigma k\frac{v_P}{v_R}C_O$。

由此可得最终 D-CACHEJOIN 算法的整体吞吐率为

$$\mu=\frac{\lambda}{T_{\mathrm{cost}}} \tag{1}$$

用 n_{SP} 表示流探测阶段中每次迭代处理的元组数量，n_{DP} 表示磁盘探测阶段中每次迭代处理的元组数量，则算法的吞吐率还可以表示为

$$\mu=\frac{n_{\mathrm{SP}}+n_{\mathrm{DP}}}{T_{\mathrm{cost}}} \tag{2}$$

式(1)和式(2)能否精确表示流数据和关系表进行连接操作的开销的前提是算法必须只需要有限的内存就可以持续运行，同时流数据 S 中新到达的元组进入系统时，需要有剩余的缓冲区内存对其进行保存。更加详细的关于关系表 R 的分页数和流缓冲区中元组个数的理论论证可参考文献[6]。

2.2　算法成本模型

由式(2)可知，算法的吞吐率与 R 的设置情况、

S 的设置情况、流和磁盘缓冲区等因素有关，本节通过定量计算来描述算法的成本。

1)　n_{SP} 的模型计算

在流检测阶段中，S 加载到流缓冲区后立即与 H_R 中的元组进行匹配，且优先与 H_R 中较多次与 R 中元组成功匹配的元组进行，如果匹配成功，则直接生成结果输出；如果匹配失败，则该元组在后续的磁盘检测阶段尝试进行匹配。由此，n_{SP} 与 H_R、R 中元组大小、R 中元组个数、流缓冲区大小有关，其中流缓冲区大小在后续实验中得知极小（始终在 0.1 MB 以下），因此忽略不计。

根据表 1，n_R 为 R 中元组个数，$n_{h_R}=b\dfrac{v_P}{v_R}$ 为 H_R 中元组个数。现实中在构建 R 和 S 时，由于其数据分布并不均匀，需要考虑其扭曲分布系数 Zipf，如果 Zipf 值为 0，那么连接属性值完全均匀分布，随着 Zipf 值的增大，数据的不均匀分布程度就越大。假设流数据 S 有 $S\sim\mathrm{Zipfian}(1,n)$，其概率密度函数为 $f(x)=\dfrac{1}{xH_{n,1}}(x=1,2,\cdots,n)$，累积分布函数为 $F(x)=\dfrac{H_{x,1}}{H_{n,1}}(x=1,2,\cdots,n)$，其中 $H_{n,1}=\sum\limits_{i=1}^{n}\dfrac{1}{i}$，$H_{x,1}$ 同理[20]。通过归一化处理，流检测阶段的匹配概率 p_1 可表示为

$$p_1=\frac{\sum\limits_{x=1}^{n_{h_R}}\frac{1}{x}}{\sum\limits_{x=1}^{n_R}\frac{1}{x}}=\frac{\ln n_{h_R}+\gamma+\varepsilon_{n_{h_R}}}{\ln n_R+\gamma+\varepsilon_{n_R}}\sim\frac{\ln n_{h_R}}{\ln n_R}\tag{3}$$

其中，$\gamma\approx0.577\,215\,6\cdots$ 为欧拉常数，$\lim\limits_{n\to\infty}\varepsilon_n=0$，二者可忽略；$\sim$ 表示近似于。故 p_1 可以用 n_{h_R} 和 n_R 表示。现考察 n_{h_R} 和 n_R 分别变为 2 倍时 p_1 的变化情况。假设此时 p_1 分别增大一个常数因子 φ_1 和减小一个常数因子 ψ_1，且有

$$p_1=n_{h_R}^{x}n_R^{y}\tag{4}$$

其中，x 和 y 是未知数。

当 n_{h_R} 变为原来的 2 倍时，式(4)变为

$$\varphi_1p_1=(2n_{h_R})^{x}n_R^{y}\tag{5}$$

将式(5)代入式(4)，得 $\varphi_1=2^{x}$，即 $x=\mathrm{lb}\,\varphi_1$。

当 n_R 变为原来的 2 倍时，式(4)变为

$$\psi_1p_1=n_{h_R}^{x}(2n_R)^{y}\tag{6}$$

将式(6)代入式(4)，得 $\psi_1=2^{y}$，即 $y=\mathrm{lb}\,\psi_1$。

因此，式(4)变为

$$p_1=n_{h_R}^{\mathrm{lb}\,\varphi_1}n_R^{\mathrm{lb}\,\psi_1}\tag{7}$$

若在 n 次迭代中算法处理的流元组总量为 S_n，则有

$$n_{\mathrm{SP}}=p_1\frac{S_n}{n}=\frac{n_{h_R}^{\mathrm{lb}\,\varphi_1}n_R^{\mathrm{lb}\,\psi_1}S_n}{n}\tag{8}$$

2)　n_{DP} 的模型计算

在流检测阶段，算法已经通过 H_R 连接匹配了 S 中的部分元组，在磁盘检测阶段中，将使用常规方式进行关系表 R 和流数据 S 中元组的连接。由于 R 较大，算法分段对 R 进行读取，每次读取磁盘缓冲区大小 n_{RB} 的数据段，由于 $S\sim\mathrm{Zipfian}(1,n)$，因此可以逐段计算匹配概率后相加，表示为

$$\sum_{x=n_{h_R}+1}^{n_{h_R}+n_{\mathrm{RB}}}\frac{1}{x}+\sum_{x=n_{h_R}+n_{\mathrm{RB}}+1}^{n_{h_R}+2n_{\mathrm{RB}}}\frac{1}{x}+\sum_{x=n_{h_R}+2n_{\mathrm{RB}}+1}^{n_{h_R}+3n_{\mathrm{RB}}}\frac{1}{x}+\cdots+\sum_{x=n_{h_R}+(n-1)n_{\mathrm{RB}}+1}^{n_{h_R}+nn_{\mathrm{RB}}}\frac{1}{x}\tag{9}$$

化简得 $\sum\limits_{x=n_{h_R}+1}^{n_{h_R}+nn_{\mathrm{RB}}}\dfrac{1}{x}$，即 $\sum\limits_{x=n_{h_R}+1}^{n_R}\dfrac{1}{x}$。

记 R 中元组的分段数为 N，由匹配概率之和经归一化后可得平均匹配概率 p_2 为

$$p_2=\frac{\sum\limits_{x=n_{h_R}+1}^{n_R}\frac{1}{x}}{N\sum\limits_{x=1}^{n_R}\frac{1}{x}}\sim\frac{\ln n_R-\ln n_{h_R}}{N\ln n_R}\tag{10}$$

类似于 n_{SP} 的计算方式，p_2 可以用 n_{RB}、n_R 和 n_{h_R} 表示$\left(N=\dfrac{n_R}{n_{\mathrm{RB}}}\right)$。现考察 n_{RB}、n_{h_R} 和 n_R 分别变为 2 倍时 p_2 的变化情况。假设此时 p_2 分别增大一个常数因子 θ_2、增大一个常数因子 φ_2 和减小一个常数因子 ψ_2，且有

$$p_2=n_{\mathrm{RB}}^{z}n_{h_R}^{x}n_R^{y}\tag{11}$$

和 n_{SP} 中处理方法相同，此时可得 $x=\mathrm{lb}\,\varphi_2$，$y=\mathrm{lb}\,\psi_2$，$z=\mathrm{lb}\,\theta_2$。

因此，式(11)可以变为

$$p_2 = n_{\mathrm{RB}}^{\mathrm{lb}\theta_2} n_{h_R}^{\mathrm{lb}\varphi_2} n_R^{\mathrm{lb}\psi_2} \tag{12}$$

若哈希表中存储的流元组总量为 S_n'，则有

$$n_{\mathrm{DP}} = p_2 S_n' = n_{\mathrm{RB}}^{\mathrm{lb}\theta_2} n_{h_R}^{\mathrm{lb}\varphi_2} n_R^{\mathrm{lb}\psi_2} S_n' \tag{13}$$

确定 n_{SP} 和 n_{DP} 后，即可通过式(2)对算法进行调整。

2.3 实验设计

实验环境为 4 台 IBM POWER x-236 8841 服务器，Intel Xeon Gold 6248R CPU，64 GB 内存，Linux 64 位操作系统，具体配置情况如表 2 所示。

表 2 实验环境

配置	值
服务器名称	IBM POWER x-236 8841
CPU 型号	Intel Xeon Gold 6248R
CPU 核心数	2
CPU 主频	3.0 GHz
内存	64 GB
操作系统	CentOS 7 64 位
Hadoop	Hadoop-2.6
硬盘	Samsung 850 PRO 256 GB SSD
JDK	JDK-8u201

环境部署的规模主要从数据访问峰值、吞吐量要求和响应时间等综合考量。基本思路是读写分离。通过解析查询语句，对于仅包括读操作的配置一个连接字符串到读服务器，对于写操作则配置另一个连接字符串到写服务器，将大规模数据的访问分流到多台服务器上，使应用中读取数据的速度和并发量显著提高，增加了系统响应速度，减少了服务器的压力，能够有效增加系统的稳定性和扩展性。在读写分离的基础上再进行负载均衡，具体过程是各节点通过专用网络进行连接，对每个服务器进行监测，获取资源占用情况，整个集群可以视为一台具有超高性能的独立服务器。

实验数据来自某运营商内部的原始话单数据，以此构建流数据 S 和关系表 R，各包含约 2 000 万条数据。

由于本文算法具有缓存模块和自动控制缓存中元组换入换出的阈值的功能，因此需要一个热身阶段。由于在实际场景中，程序开始后会持续地运行，因此热身阶段是可以接受的，后续实验中如果没有特殊说明，该系统热身阶段都会忽略。

本节设计实验测试了本文算法的性能，实验的具体软硬件条件已在表 2 中列出，在此环境上搭建了具有 8 个虚拟节点的分布式集群。通过将本文算法与传统的 MESHJOIN 算法、CACHEJOIN 算法和较先进的 Eunomia 分布式连接方法进行比较，对算法的执行效率进行实验考察，各实验运行 5 次后取平均值作为实验结果。

2.4 实验结果及分析

实验 1 不同数据量对算法效率的影响

首先令集群开启 4 个节点，处理不同数量级的数据，吞吐率对比如图 4 所示，D-CACHEJOIN 都取得了比其他 3 种算法更好的吞吐率，这一方面是因为实验数据并不是均匀分布，而是服从 Zipfian 分布的，其中一部分数据具有比其他数据更多的出现次数，没有对此进行额外考虑的算法无法适应该环境；另一方面是因为 D-CACHEJOIN 具有缓存模块，能够较好地处理数据倾斜的情况。同时，D-CACHEJOIN 的吞吐率在 10 万、100 万和 1 000 万条数据的情况下分别是 Eunomia 的 1.30 倍、1.48 倍和 1.65 倍，算法的优势随着数据量的增多而逐渐增大，这是因为数据量增多时，可能对磁盘的 I/O 次数逐渐增大，D-CACHEJOIN 越来越多地减少了实际上对于磁盘的 I/O 次数。

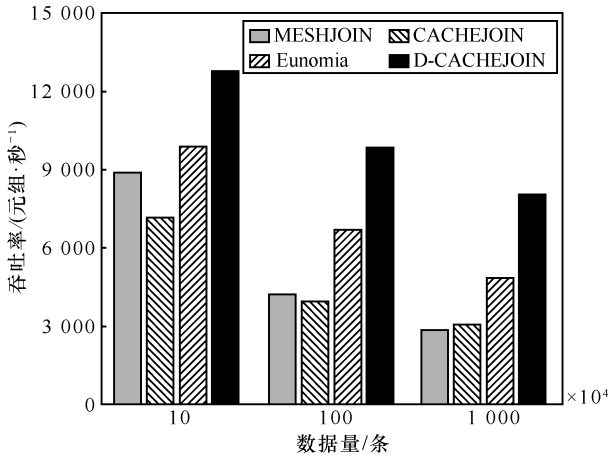


图 4 不同数据量下算法的吞吐率

现实中大多数的大型数据集是服从 Zipfian 分布的[21]，包括本文实验的数据集。相对于均匀分布的数据集，对服从 Zipfian 分布的数据集进行实验有更大的实际意义，但为了实验的完整性，给出其他实验条件相同时对均匀数据的实验结果，如图 5 所示。

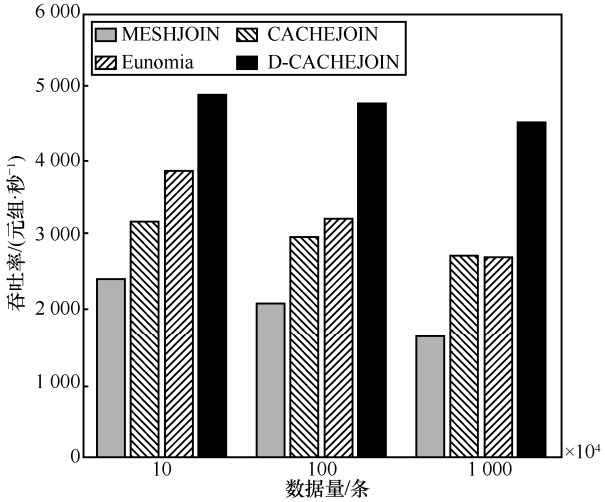


图 5　不同数据量下算法的吞吐率（均匀分布）

实验 2　可用内存总量对算法效率的影响

本文实验中可用内存的总量为磁盘缓冲区、流缓冲区、队列、H_R 和 H_S 的大小之和，但本文实验中有两点不变：流缓冲区基本不变而且可以忽略，因为在所有情况下流缓冲区使用的内存不超过 0.1 MB；H_R 保持不变，因为 H_R 的大小会在很大程度上影响本文算法的效率，所以在后续实验中专门对 H_R 的大小对算法效率的影响进行研究，故在本文实验中 H_R 保持为 R 的 1%，一旦 H_R 满，如果再次有元组的匹配频率超过 thresholdValue，将可能会对 H_R 中已有元组进行替换并可能引起 thresholdValue 的动态调整。可用内存总量对算法效率的影响如图 6 所示。

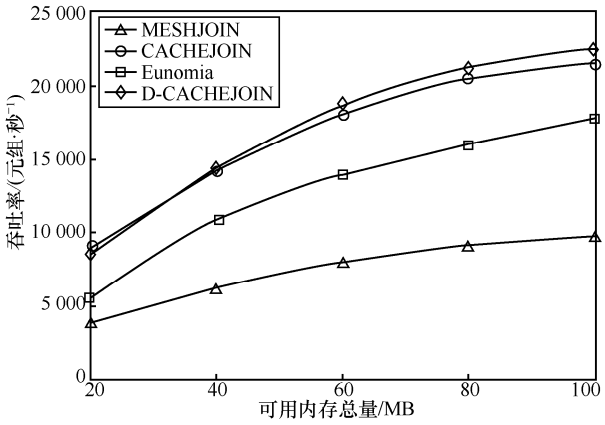


图 6　可用内存总量对算法效率的影响

从图 6 可以看出，几种流连接算法的吞吐率都会随着可用内存总量的增加而提高，由于引入了缓存模块，CACHEJOIN 和 D-CACHEJOIN 的吞吐率都要比 MESHJOIN 高 2 倍以上，而 Eunomia 由于需要基于时间戳对流数据进行检查，因此需要较多的额外内存才能表现出较高的效率，内存较少时执行效率不足。另有其他实验显示，当提供大量内存时，Eunomia 的吞吐率已经超过了 CACHEJOIN，但仍低于 D-CACHEJOIN。考虑到实验的完整性，补充了内存较少时的情况，由于 D-CACHEJOIN 需要存储哈希函数的计算结果以便后续使用，因此当可用内存较少时，D-CACHEJOIN 的吞吐率要略低于 CACHEJOIN。由于本文研究的支撑系统具有充足的内存可供使用，同时鉴于现代分布式系统中如果有计算需要，通常都会配备充足的内存，因此一般不会出现内存严重不足的情况。当可用内存增大后，D-CACHEJOIN 的吞吐率逐渐超过 CACHEJOIN，且随着内存的增加，前者吞吐率的增加速度要高于后者；当内存达到 100 MB 时，D-CACHEJOIN 的吞吐率相比于 CACHEJOIN 增加了 10%以上。

实验 3　R 的缓存比例对算法效率的影响

由于算法引入了缓存模块，因此可对 R 中频繁匹配的元组进行缓存，能够大大提升算法的效率，本文实验固定可用内存大小，研究 R 的缓存比例对算法效率的影响，实验结果如图 7 所示。

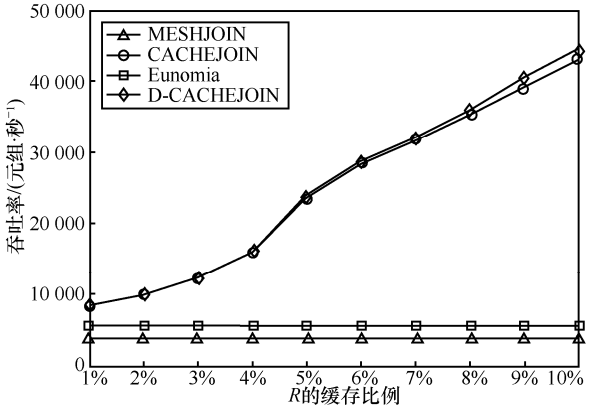


图 7　R 的缓存比例对算法效率的影响

根据图 7 可以看出，当 R 的缓存比例达到 10% 时，D-CACHEJOIN 算法的吞吐率是 MESHJOIN 算法的 10 倍以上。当缓存比例较低时，D-CACHEJOIN 算法的吞吐率仍然是 MESHJOIN 算法的 2 倍以上，由于没有引入缓存策略，MESHJOIN 和 Eunomia 的吞吐率保持不变。这说明引入缓存模块可以大大提升算法的性能。

实验 4　D-CACHEJOIN 算法在分布式集群中的扩展性

为了测试算法在分布式集群中的扩展情况，将集

群中的节点数 n 由 4 增加到 8，并且改变算法处理的数据量，考察此时的执行效率，实验结果如图 8 所示。

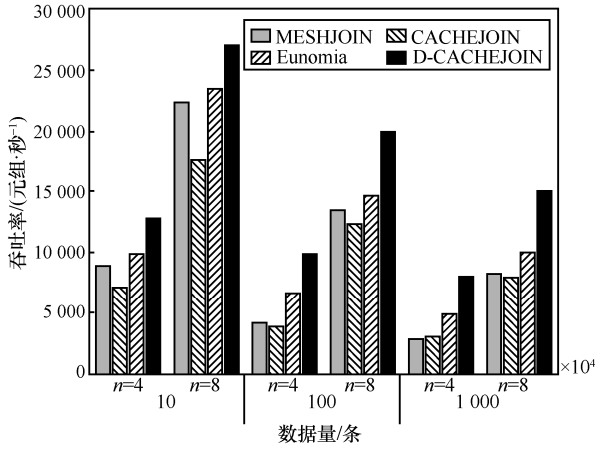


图 8 算法吞吐率随节点扩展的变化情况

总体而言，各算法的吞吐率随着节点数的增多而增大，且 D-CACHEJOIN 保持着最佳的执行效率。图 8 中从左至右，D-CACHEJOIN 的吞吐率依次是 Eunomia 的 1.29 倍、1.15 倍、1.48 倍、1.36 倍、1.65 倍、1.50 倍，这表示随着集群中节点数的增加，D-CACHEJOIN 相对于 Eunomia 在执行效率上的优势变小了（对其他 2 种算法也有这一结论）。这是因为随着节点数的增多，每个节点上关系表 R 的片段就越小，用于磁盘 I/O 的时间就越短，而 D-CACHEJOIN 恰恰减少了磁盘 I/O 的次数，因此这一优势在节点数增加时变小了。但与对内存的访问相比，磁盘 I/O 是非常消耗时间的操作，因此即使集群进一步扩容，只要磁盘上的关系表无法全部放入内存中，D-CACHEJOIN 算法仍然会有更优的执行效率。

综上，D-CACHEJOIN 算法总体上保持了最优的执行效率。当 ETL 系统中数据的数量级发生比较大的变化时，由于对数据库的访问时间，即对磁盘的 I/O 所需时间会随之增大，算法的吞吐率会随之下降，但本文算法的优势也会逐渐增大，因为本文算法减少了对磁盘的 I/O 次数。为了拥有更高的效率，算法往往会进行更多消耗内存的额外工作，在现代处理海量数据的分布式系统中，通常都会配备足够的内存，同时本文算法拥有一定的自由度，可以根据实际情况设置缓存的大小，更好地贴合具体场景，本文算法一般能够很好地适应环境并良好运行。当分布式集群中新增或下线节点时，本文算法保持了更优的执行效率；当处理海量数据时，本文算法在分布式集群中具有良好的拓展性。

3 结束语

本文介绍了流数据接收过程中面临的主要问题，并简单介绍了已有的流连接算法的处理策略，在此基础上对算法进行改进，提出了一种把使用缓存的流连接策略应用于分布式环境的 D-CACHEJOIN 算法。此外，本文详细描述了该算法的执行架构并计算其性能开销，通过实验展示了该算法的执行效率，说明了该算法具有较好的适应性，在一般的大数据处理系统中能够良好运行，并且在分布式集群中具有良好的拓展性。

参考文献：

[1] POLYZOTIS N, SKIADOPOULOS S, VASSILIADIS P, et al. Supporting streaming updates in an active data warehouse[C]//Proceedings of 2007 IEEE 23rd International Conference on Data Engineering. Piscataway: IEEE Press, 2007: 476-485.

[2] 林子雨, 林琛, 冯少荣, 等. MESHJOIN*: 实时数据仓库环境下的数据流更新算法[J]. 计算机科学与探索, 2010, 4(10): 927-939.
LIN Z Y, LIN C, FENG S R, et al. MESHJOIN*: an algorithm supporting streaming updates in a real-time data warehouse[J]. Journal of Frontiers of Computer Science & Technology, 2010, 4(10): 927-939.

[3] KIM H J, LEE K H. Semi-stream similarity join processing in a distributed environment[J]. IEEE Access, 2020, 8: 130194-130204.

[4] 熊超. 多路数据流等值连接中独立元素问题的研究[D]. 深圳: 中国科学院大学(中国科学院深圳先进技术研究院), 2020.
XIONG C. The distinct element problem in equi-join for multiple data streams[D]. Shenzhen: Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, 2020.

[5] 魏星贝, 李陶深, 许嘉, 等. QJoin：质量驱动的乱序数据流连接处理技术[J]. 广西科学, 2020, 27(3): 266-275.
WEI X B, LI T S, XU J, et al. QJoin: quality-driven join processing technique over out-of-order data streams[J]. Guangxi Sciences, 2020, 27(3): 266-275.

[6] POLYZOTIS N, SKIADOPOULOS S, VASSILIADIS P, et al. Meshing streaming updates with persistent data in an active data warehouse[J]. IEEE Transactions on Knowledge and Data Engineering, 2008, 20(7): 976-991.

[7] ZULFIKAR A F, LESLIE HENDRIC SPITS WARNAS H, GAOL F L, et al. Query optimization for distributed databases uses a semi-join based approach (SBA) with the SDD-1 algorithm[C]//Proceedings of 2019 International Conference on Information and Communications Technology (ICOIACT). Piscataway: IEEE Press, 2019: 619-623.

[8] VAIDEHI V, DEVI D S. Distributed database management and join of multiple data streams in wireless sensor network using querying techniques[C]//Proceedings of 2011 International Conference on Recent Trends in Information Technology (ICRTIT). Piscataway: IEEE Press, 2011: 583-588.

[9] 陈付梅, 韩德志, 毕坤, 等. 大数据环境下的分布式数据流处理关键技术探析[J]. 计算机应用, 2017, 37(3): 620-627.
CHEN F M, HAN D Z, BI K, et al. Key technologies of distributed data stream processing based on big data[J]. Journal of Computer Applications, 2017, 37(3): 620-627.

[10] ABDELHAFIZ B M. Distributed database using sharding database architecture[C]//Proceedings of 2020 IEEE Asia-Pacific Conference on Computer Science and Data Engineering. Piscataway: IEEE Press, 2020: 1-17.

[11] NAEEM M A, DOBBIE G, WEBER G. A lightweight stream-based join with limited resource consumption[C]//Proceedings of the 14th international conference on Data Warehousing and Knowledge Discovery. Berlin: Springer, 2012: 431-442.

[12] JEON Y H, LEE K H, KIM H J. Distributed join processing between streaming and stored big data under the micro-batch model[J]. IEEE Access, 2019, 7: 34583-34598.

[13] YUAN J, WANG Y H, CHEN H H, et al. Eunomia: efficiently eliminating abnormal results in distributed stream join systems[C]//Proceedings of 2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS). Piscataway: IEEE Press, 2021: 1-11.

[14] 狄程, 杨中国, 韩燕波, 等. 面向流数据的实时处理及服务化系统[J]. 重庆大学学报, 2020, 43(7): 75-83.
DI C, YANG Z G, HAN Y B, et al. View-driven flow data oriented real-time processing and service system[J]. Journal of Chongqing University, 2020, 43(7): 75-83.

[15] DAVID K, KEHMAN E, LEIGHTON T, et al. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web [C]//Proceedings of the 29th ACM Symposium on Theory of Computing. New York: ACM Press, 1997: 654-663.

[16] KNUTH D E. The art of computer programming, vol. 3: sorting and searching, 2nd edition[M]. Redwood City: Addison Wesley Longman Publishing Co., Inc., 1998.

[17] ARMSTRONG R. The long tail: why the future of business is selling less of more[J]. Canadian Journal of Communication, 2008, 33(1): 274-276.

[18] WORRELL J. Real-time model checking: algorithms and complexity[C]//Proceedings of 2008 15th International Symposium on Temporal Representation and Reasoning. Piscataway: IEEE Press, 2008: 19.

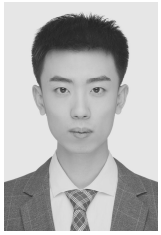
[19] NAEEM M A, BAJWA I S, JAMIL N. A cached-based approach to enrich Stream data with master data[C]//Proceedings of 2015 Tenth International Conference on Digital Information Management (ICDIM). Piscataway: IEEE Press, 2015: 57-62.

[20] NIAN H, CHEN L, XU Y Y, et al. Sequences domain impedance modeling of three-phase grid-connected converter using harmonic transfer matrices[J]. IEEE Transactions on Energy Conversion, 2018, 33(2): 627-638.

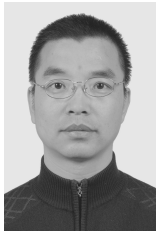
[21] MOTWANI R, VASSILVITSKII S. Distinct values estimators for power law distributions[C]//Proceedings of the 3rd Workshop on Analytic Algorithmics and Combinatorics (ANALCO). Philadelphia: Society for Industrial and Applied Mathematics, 2006: 230-237.

[作者简介]

张笑燕（1973− ），女，山东烟台人，博士，北京邮电大学教授，主要研究方向为软件工程理论、移动互联网软件与大数据分析。



刘志浩（1996− ），男，山东临沂人，北京邮电大学硕士生，主要研究方向为大数据分析、移动与互联网软件。



杜晓峰（1973− ），男，陕西韩城人，北京邮电大学讲师，主要研究方向为云计算与大数据分析。

陆天波（1977− ），男，贵州毕节人，博士，北京邮电大学教授，主要研究方向为网络与信息安全、安全软件工程和 P2P 计算。