针对高速数据流的大规模数据实时处理方法

亓开元[1),2)] 赵卓峰[1),3)] 房 俊[1),3)] 马 强[1),2)]

[1)](中国科学院计算技术研究所 北京 100190)

[2)](中国科学院研究生院 北京 100190)

[3)](北方工业大学信息工程学院 北京 100144)

摘 要 以实时传感数据和历史感知数据为基础的各类计算需求逐渐成为当前物联网应用建设中的关键,如何实现基于高速数据流和大规模历史数据的实时计算成为数据处理领域的新挑战.现有批处理方式的MapReduce大规模数据处理技术难以满足此类计算的实时要求.文中结合城市车辆数据的实时采集与处理应用,在理论和实践分析的基础上,提出了一种针对高速数据流的大规模数据实时处理方法,并对方法中的本地阶段化流水线、中间结果缓存等关键技术瓶颈进行了改进.其中,根据系统参数控制阶段化流水线,使CPU得到了充分、有效利用;通过改造内外存数据结构、读写策略和替换算法,优化了本地中间结果的高并发读写性能.实验表明,上述方法可以显著提升大规模历史数据上数据流处理的实时性和可伸缩性.

关键词 数据流处理;大规模数据处理;MapReduce;物联网;大数据;云计算

中图法分类号 TP393 **DOI号**:10.3724/SP.J.1016.2012.00477

Real-Time Processing for High Speed Data Stream over Large Scale Data

QI Kai-Yuan[1),2)] ZHAO Zhuo-Feng[1),3)] FANG Jun[1),3)] MA Qiang[1),2)]

[1)](*Institute of Computing Technology, Chinese Academy of Sciences, Beijing* 100190)

[2)](*Graduate University of Chinese Academy of Sciences, Beijing* 100190)

[3)](*College of Information Engineering, North China University of Technology, Beijing* 100144)

Abstract With the development of Internet of Things, the computing based on real-time and historical sensor data becomes the key point to the IoT applications, and how to support the real-time processing for high speed data stream over large scale data brings a new challenge. However, the existing large scale data processing technology based on the MapReduce model is designed for batch processing and cannot satisfy the real-time requirement. Based on the theory and practice analysis, this paper proposes a method for large scale data processing under high speed data stream, and improves the technical bottlenecks such as local staged pipeline and intermediate result storage. We tune the configuration of staged pipeline dynamically using system information to efficiently utilize CPU, and design the data structure, read/write operation strategy and replacement algorithm to optimize the high concurrency access performance of local intermediate results. The experiment shows that this method can improve real-time performance and scalability of data stream processing over large scale history data.

Keywords data stream processing; large scale data processing; MapReduce; Internet of Things; big data; cloud computing

---

收稿日期:2011-05-25;最终修改稿收到日期:2012-01-26.本课题得到国家自然科学基金(60903137,61003294)资助.**亓开元**,男,1984年生,博士研究生,主要研究方向为大规模数据处理、服务计算.E-mail:7hyl@163.com.**赵卓峰**,男,1977年生,博士,副研级高工,主要研究方向为云计算、大规模数据处理、服务计算.**房 俊**,男,1976年生,博士,助理研究员,主要研究方向为服务计算、云计算.**马 强**,男,1986年生,硕士研究生,主要研究方向为数据流处理、大规模数据处理.

1 引 言

随着物联网的发展,以实时传感数据为基础的各类数据流处理逐渐成为当前物联网应用构造的关键.面对持续到达的数据流,数据流处理系统必须快速对其进行响应并即时输出结果.由于数据流的连续性和无限性,有限的处理机不可能处理数据流的完整信息,因而采用窗口机制(时间、数据量)来划定处理边界,窗口边界范围内已积累的数据称之为历史数据.传统数据流处理[1-3]受数据采集速度、传输带宽和内存容量等因素的限制,侧重于针对一个相对小的历史数据规模进行.随着数据采集和传输技术的进步,使得短时间内积累大量历史数据成为可能.同时,当前物联网环境下数据流处理应用的长期性、全面性和准确性需求也要求扩大历史数据规模.以一个物联网环境下的车辆监管应用为例.该应用通过传感设备对城市车辆实时数据(包括道路运行车辆和停车场静止车辆)进行收集,并在已收集数据的基础上进行套牌、超速、限行等多种违法车辆的自动识别计算.应用面对的数据一方面是高速到达的数据流,另一方面是持久化的历史数据,要求实时的完成数据流同历史数据的分析、比较等计算.类似的应用还包括网络入侵检测、Web 个性化搜索等.在这类应用中,由于窗口范围的不断变大,数据处理对象(如车牌)数量的急速增加,以及每个数据对象数据量(如车辆监控信息)的迅速增加,造成了历史数据规模不断扩大.在上述趋势下,面向大规模历史数据的数据流实时处理需求同计算、存储能力不足之间的矛盾成为云计算和物联网领域的新挑战.本文将此问题定义为数据流处理的可伸缩性问题.

现有数据流处理的可伸缩性研究可分为集中式和分布式两类.在集中式环境下,受计算和存储(主要是内存)资源限制,主要通过概要数据[4]、准入控制[1]、QoS 降阶[2]等方法,以牺牲服务质量为代价保障伸缩性.在分布式环境下,针对由多个处理算子组成的数据流处理网络,主要通过在多个节点上平衡算子的分布来保障伸缩性[2-3],但处理能力仍局限于单个算子所在节点所能处理的数据窗口,在面对大规模历史数据的情况下伸缩能力不足.

面向大规模历史数据的数据流处理需要突破单个节点的内存和计算能力限制.计算、存储设备性能的提高、成本的降低和大规模数据处理技术的不断成熟,为采用无共享集群架构解决此类数据处理问题创造了条件.为了支撑大规模数据的存储和计算,当前往往采用具有多个 CPU 的多核集群计算架构,以及 Cache、内存、外存和分布式存储四层存储结构,如图 1.在这种架构中,节点上的多核 CPU 构成了本地计算资源;相比于分布式存储,节点上的内存和外存组成了高速的本地存储.在无共享集群架构下,利用 MapReduce[5]编程模型解决大规模数据处理需求同计算、存储能力不足之间的矛盾是云计算的核心技术,MapReduce 通过简单的编程接口为并行处理可划分的大规模数据提供了支持,向程序员屏蔽了任务调度、数据存储和传输等细节,非常适合解决数据处理问题的伸缩性需求.然而,现有的 MapReduce 方法,如 Hadoop①、Phoenix[6]等,属于对持久化数据的批处理方式,在每次处理时,都需要初始化运行环境,重复载入、处理大规模数据,同步执行 Map 和 Reduce 阶段,并在节点间传递大量数据.以批处理方式处理持续到达的数据流,若每次处理小规模批的数据,则系统开销太大,实时性受到限制,若等待批达到一定规模又增加了处理延迟,同样无法满足实时需求.因此,针对高速数据流下的大规模数据实时处理需求,如何利用 MapReduce 模型是需要考虑的问题.

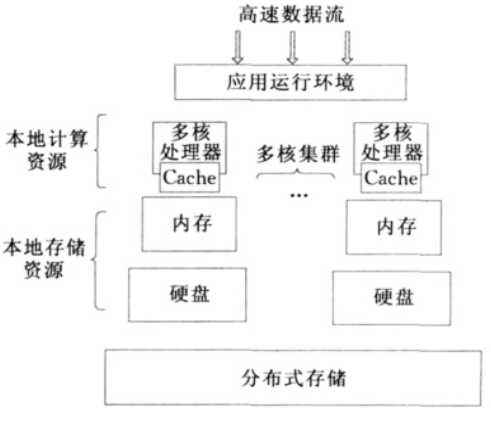

图 1 支撑环境

为增强 MapReduce 的数据流处理能力,可以通过预处理、分布缓存和复用中间结果的方法避免每次数据流到达时的历史数据重复处理开销,并使得数据流处理本地化,减少节点间的数据传输开销.针对本地化的数据流处理,可以采用事件驱动的阶段化处理架构[7](Staged Event Driven Architecture,

① Apache Hadoop [EB/OL]. http://hadoop.apache.org/ 2011,8,17

SEDA)，利用线程池技术减少每次处理的初始化开销，并通过划分阶段和在阶段间异步传递数据，消除阶段之间的数据同步.

基于以上思想，本文首先通过理论和实践分析证明了采用 MapReduce 模型解决此类问题的适应性，并在此基础上提出了一种支持高速数据流下大规模数据实时处理的方法 RTMR(Real-Time MapReduce). RTMR 的处理过程为预处理历史数据并将中间结果分布缓存到各个节点上，在节点上基于 SEDA 构造从 Map 阶段到 Reduce 阶段的本地阶段化流水线，充分利用本地计算和存储资源实现数据流同历史数据的实时计算. 在上述方法中，还存在以下挑战性问题：

(1) Esper① 公布的数据表明，CPU 是影响数据流处理性能的关键资源. 在大规模历史数据情况下，计算量骤增，使得有效利用 CPU 更为重要. 因此，如何充分、有效地利用 CPU(包括 CPU Cache)提高实时处理能力，是本地阶段化处理面临的关键问题.

(2) 在本地阶段化架构下，针对 Map 和 Reduce 线程对中间结果的频繁读写，应该如何支持对中间结果本地存储的高并发访问也是需要解决的问题.

针对上述问题，RTMR 方法包括了一种基于系统参数的本地阶段化处理优化方法和支持高并发读写的本地存储方法.

2　针对高速数据流的大规模数据实时处理方法

批处理方式的 MapReduce 无法满足物联网环境下数据流的实时处理需求. 本节在理论和实践分析的基础上，基于 MapReduce 模型提出了支持高速数据流下大规模数据实时处理的方法 RTMR.

2.1　RTMR 方法

MapReduce 模型的定义[5]为

$$\text{Map}: k_1, v_1 \rightarrow list(k_2, v_2);$$

$$\text{Reduce}: k_2, list(v_2) \rightarrow list(v_2).$$

其处理过程是，Map 方法将$[k_1, v_1]$键值对转换为$[k_2, v_2]$键值对，Reduce 方法针对每个 k_2 的值列表 $list(v_2)$ 做 *list* 操作. 将上述模型改造为函数形式，若待处理数据为 D，Map 阶段中间结果为 I，MapReduce过程可以表示为

$$MR(D)=R(M(D))=list(I),$$

其中，M 表示 Map 方法，R 表示 Reduce 方法，*list* 表示 Reduce 方法所做的操作. 下面分析 MapReduce 模型的性质.

定义 1.　对于函数 $F: I \rightarrow O$，若存在函数 $P: O \times O \rightarrow O$，使得 $F(D+\Delta)=P(F(D), F(\Delta))$，则称 F 为可合并的.

定义 2.　对于数据集 D 的 n 个数据子集 $D_1, D_2, \cdots, D_n$，若 $D_1 \cap D_2 \cap \cdots \cap D_n = \varnothing$ 并且 $D_1 \cup D_2 \cup \cdots \cup D_n = D$，则称 $D_1, D_2, \cdots, D_n$ 为 D 上的一个划分.

定义 3.　对于键值对数据集合 D，键集合 K，称集合$\{d \mid d.key \in K, d \in D\}$为 D 在 K 上的投影，记为 $\sigma_K(D)$.

由MapReduce模型和上述定义可知，MapReduce具有以下性质：

(1) Map 方法满足分配率. 即两个数据集合并集上的 Map 等于分别对两个集合 Map 的并集

$$M(D+\Delta)=M(D)+M(\Delta).$$

(2) Reduce 方法具有可合并性. 即

$$list(D+\Delta)=list(list(D), list(\Delta)).$$

(3) Reduce 方法满足分配率，即若 $K_1, K_2, \cdots, K_n$ 为中间结果 I 键集合的一个划分，则

$$list(I)=list(\sigma_{K_1}(I))+list(\sigma_{K_2}(I))+\cdots+list(\sigma_{K_n}(I)).$$

定理 1.　MapReduce 为可合并的.

证明.　根据 MapReduce 的性质，对于数据 D 和增量 Δ 有

$$\begin{aligned} &MR(D+\Delta)\\ &=R(M(D+\Delta))\\ &=R(M(D)+M(\Delta))\\ &=list(I_D+I_\Delta)\\ &=list(list(I_D), list(I_\Delta))\\ &=R(MR(D), MR(\Delta)). \end{aligned}$$

因此，由定义 1 可知 MapReduce 是可合并的.

证毕.

定理 1 表明 MapReduce 模型可通过预处理历史数据并缓存中间结果的方法降低每次数据流到达时的重复处理开销，提高实时处理能力. 将上述过程用函数表示为

$$MR(D+\Delta)=list(I_D+I_\Delta)=MR(\Delta \mid I_D).$$

定理 2.　$K_1, K_2, \cdots, K_n$ 为 MapReduce 中间结果 I 键集合的一个划分，对于数据增量 Δ 在 I 上的 MapReduce，有

$$MR(\Delta \mid I)=MR(\Delta \mid \sigma_{K_1}(I))+MR(\Delta \mid \sigma_{K_2}(I))+\cdots+MR(\Delta \mid \sigma_{K_n}(I)).$$

① Esper performance [EB/OL]. http://docs.codehaus.org/display/ESPER/Esper+performance 2011, 8, 17

证明. 由 MapReduce 的性质,对于中间结果 I 和数据增量 Δ,有

$$MR(\Delta|I)=list(I+I_\Delta)$$
$$=list(\sigma_{K_1}(I+I_\Delta))+list(\sigma_{K_2}(I+I_\Delta))+\cdots+list(\sigma_{K_n}(I+I_\Delta)).$$

对于 Reduce 方法来说,在中间结果的 K_1 投影上处理的数据增量仅与 K_1 有关,即

$$MR(\Delta|\sigma_{K_1}(I))=list(I_\Delta+\sigma_{K_1}(I))$$
$$=list(\sigma_{K_1}(I_\Delta+\sigma_{K_1}(I)))=list(\sigma_{K_1}(I_\Delta+I)).$$

同理,

$$MR(\Delta|\sigma_{K_2}(I))=list(\sigma_{K_2}(I_\Delta+I)),$$
$$\cdots\cdots$$
$$MR(\Delta|\sigma_{K_n}(I))=list(\sigma_{K_n}(I_\Delta+I).$$

因此,

$$MR(\Delta|I)=MR(\Delta|\sigma_{K_1}(I))+MR(\Delta|\sigma_{K_2}(I))+\cdots+MR(\Delta|\sigma_{K_n}(I)).$$

证毕.

定理 2 表明 MapReduce 模型通过分布缓存中间结果,可以使数据流同中间结果的计算仅发生在节点本地.由于避免了节点间的数据传输,因此合理的分布中间结果能够保障整个架构的可伸缩性.

从实践角度,以套牌车计算为例,在 2.93 GHz CPU,2GB 内存和 1Gbps 以太网连接的服务器集群上处理数据流,在已有 100 MB 历史数据的情况下,针对每条 200 B 的数据流,从接收到完成 Hash 分组操作的平均耗时是 4(±0.5 μs),完成数据流同历史数据之间比较操作的平均耗时是 1(±0.2 ms),在服务器之间完成数据传输的平均延迟为 40(±4 μs).上述实验数据表明一颗 2.93 GHz CPU 能够以超过 50 MB/s 的速度接收和分组数据流,而在实际应用中,受采集端带宽等限制,数据流远达不到这个速度.因此,对于多核系统,接收和分组数据流(Map 阶段)只需占用很少一部分 CPU 资源,是套牌计算这类应用所要面对的次要矛盾,主要矛盾是数据流同大规模数据之间的统计、比较等计算(Reduce 阶段)以及节点之间的数据传输.为了降低处理主要矛盾的系统开销,也就是减少历史数据重复处理和避免节点间数据传输,可以采用前述理论分析论证的技术路线:将预处理的历史数据中间结果分布缓存于各节点;每个节点冗余地接收数据流,通过 Map 阶段过滤出本节点负责处理的数据并在本地缓存上进行 Reduce 计算.当已有节点的本地计算和存储资源不能满足实时性需求时,可以在新增节点上通过重新划分和移动缓存数据进行扩展.

有鉴于此,我们设计了大规模历史数据上的数据流处理方法 RTMR,如图 2.

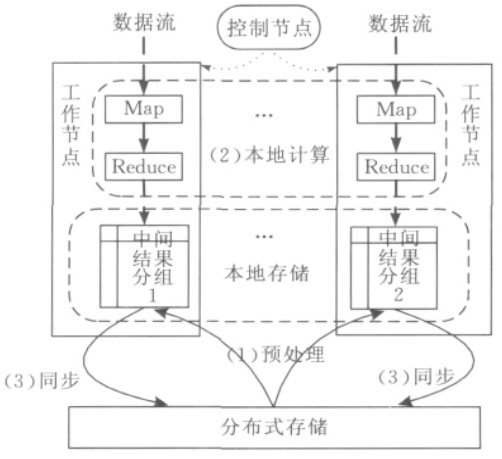

图 2 RTMR 方法

RTMR 的工作过程如下.

过程 1. RTMR 过程.

1. 中间结果缓存.

预处理相关历史数据生成中间结果,根据 k_2 的 Hash 值划分区间,分布缓存到各个工作节点的本地存储上.

2. 本地阶段化处理.

Map 阶段通过作用于 k_2 的 Hash 函数分组数据流,并按照区间划分过滤出与本地中间结果相关的数据,异步的传递给 Reduce 阶段进行与中间结果的计算.

3. 数据同步.

将本地计算结果同步到分布式存储.

在 RTMR 方法中,工作节点负责维护本地中间结果缓存和阶段化流水线.控制节点负责 RTMR 任务的生命周期管理、可靠性和可伸缩性保障.本文主要对中间结果本地存储、本地阶段化流水线等技术进行讨论,对于在控制节点中需要解决的关键问题,将在后续工作介绍.

2.2 编程接口

根据 RTMR 的处理过程,RTMR 编程接口除 map 和 reduce 方法外,还包括预加载方法 load 和数据同步方法 update,如代码 1. load 方法默认以 map 和 reduce 方法预处理相关持久化历史数据生成中间结果,程序员还可以在 load 方法中实现其他相关数据的预加载,如车辆监管应用中的套牌阈值、布控黑名单列表等. update 默认将中间结果同步到分布式存储,为了增强适应性,RTMR 支持定时和即时两种同步机制.

代码 1.　RTMR 编程接口.

```
public class FakeLicenseCarJob implements Job {
    public void load() {
        …
    }
    public void update() {
        …
    }
    public static class FakeLicenseCarMapTask
implements MapTask {
        public void map() {
            …
        }
    }
    public static class FakeLicenseCarReduceTask
implements ReduceTask {
        public void reduce() {
            …
        }
    }
}
```

3　支持高并发读写的中间结果本地存储

为减少每次数据流到达时的历史数据重复计算开销，RTMR 支持中间结果缓存. 在本地阶段化架构下，Map 和 Reduce 工作线程将对中间结果频繁地进行读写，优化中间结果的高并发读写性能是提高数据流处理能力的关键. 本节在介绍中间结果内存数据结构和外存文件结构基础上，提出了一种支持高并发读写的本地存储优化方法.

3.1　中间结果存储结构

定义 4.　在MapReduce 模型中，将$[k_2, list(v_2)]$以及 $list(v_2)$称为中间结果.

借鉴 Metis[8] 的思路，中间结果在内存中采用 Hash B^+ 树结构存储，如图 3. 在这种结构中，具有相同Hash值的 k_2 在 Hash 表的同一项中用 B^+ 树组织，$[k_2, list(v_2)]$在 B^+ 树的叶节点用链表组织，$list(v_2)$存储在 B^+ 树的叶节点. Hash B^+ 树结构具有很高的读写性能. 如果 k_2 可预测并且具有唯一的 Hash 值，则可以通过为 Hash 表分配足够的项来避免冲突和 B^+ 树查找，插入和查找操作都只有 $O(1)$ 的复杂度. 如果 k_2 没有唯一的 Hash 值或不可预测，则在 Hash 表项中维护一个 B^+ 树，插入和查找操作也只有 $O(1)+O(\log n)$的复杂度.

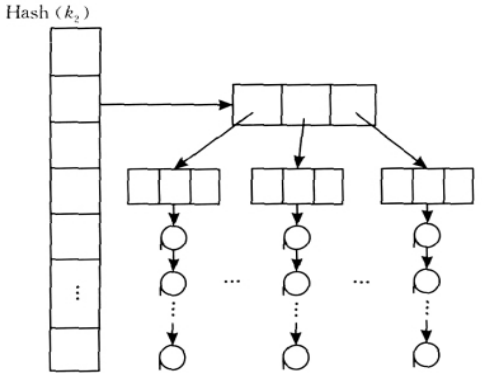

图 3　中间结果数据结构

为了扩大中间结果的本地存储容量，在外存构造 SSTable 文件[9] 存储中间结果. SSTable 文件结构包括一个索引块和多个 64 KB 的数据块（如图 4），以块为单位为 Hash 表项分配外存空间. 在数据流处理过程中，如果所需的中间结果 Hash 表项不在内存而在外存并且内存已无空间，将发生内外存替换.

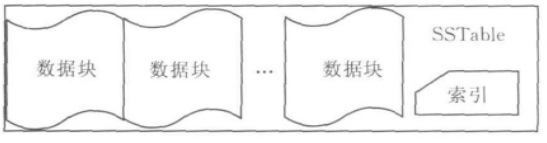

图 4　SSTable 文件结构

针对大规模的历史数据，为在集群环境下保障可伸缩，RTMR 为工作节点划分其所负责的中间结果 Hash 区间. 如图 5 所示，k_2 的 Hash 区间分布在 n 个工作节点上（$P_1, P_2, \cdots, P_n$）. 由 RTMR 过程可知，Map 阶段包括一个对于 k_2 的 Hash 分组操作，每个节点只负责区间内数据流的处理. 如果 Hash 区间划分采用对节点数取余的方法，那么在伸缩时（增减节点）会引发大量数据的移动. 为了减少伸缩时的数据移动规模，RTMR 采用一致性 Hash 算法[10] 在节点上划分中间结果. 例如，原有 P_1、P_2 和 P_3 3 个分区，当增加一个节点时，只需要将 P_1 和 P_3 的一部分划分为 P_4 即可，如图 6.

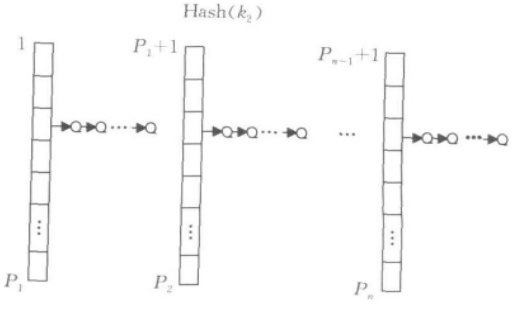

图 5　中间结果划分

在上述存储结构基础上，阶段化流水线中 Map 和 Reduce 线程对中间结果内存结构的并发读写同步，以及对中间结果外存文件的并发读写开销制约

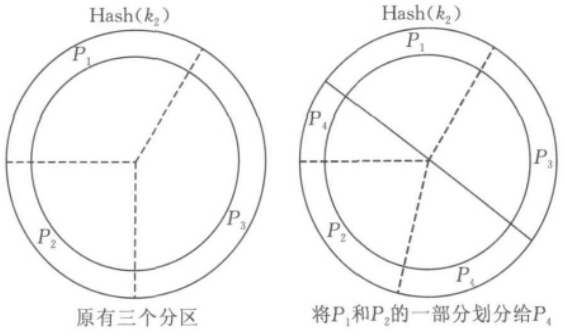

图 6 一致性 Hash 算法

了数据流实时处理能力. 为了提高本地中间结果的并发读写性能,还应从两方面进行考虑:(1)建立内存缓冲区减少并发线程之间的同步;(2)改造外存文件读写策略和替换算法降低外存的并发读写开销.

3.2 内存缓冲区

由上节可知,工作节点在内存中维护 Hash B^+ 树存放中间结果. 在本地阶段化架构下,Map 和 Reduce线程都会对中间结果频繁地进行读写,Map 线程写入处理结果,Reduce 线程读取 Map 结果,处理后写入或更新中间结果. 因此,为了提高内存并发读写性能,关键是减少 Map 和 Reduce 工作线程之间对同一块内存区域的并发读写同步开销. 通过建立 Map 和 Reduce 阶段之间的缓冲区能够避免对中间结果的读写同步.

为了避免多个 Map 线程之间对缓冲区的写同步,每个 Map 线程独占一个缓冲区. 为了避免 Map 和 Reduce 线程之间对缓冲区的读写同步,缓冲区设计成一个 FIFO 队列,Map 线程向队列尾部写入,Reduce 线程从队列头部读出. 根据 Reduce 的分配率,每个 Reduce 线程负责一个 Hash 区间,Reduce 线程之间不存在同步关系. Map 和 Reduce 线程之间对应 Hash 区间建立缓冲区,Map 线程在数据流中过滤出本节点负责的数据并将其写入相应区间的缓冲区,Reduce 线程处理负责区间内的缓冲数据并将计算结果写到 Hash B^+ 树中. 按照上述设计,根据 Map 和 Reduce 线程数量,在内存中的缓冲数据形成一个 $M\times N$ 的缓冲区矩阵,如图 7.

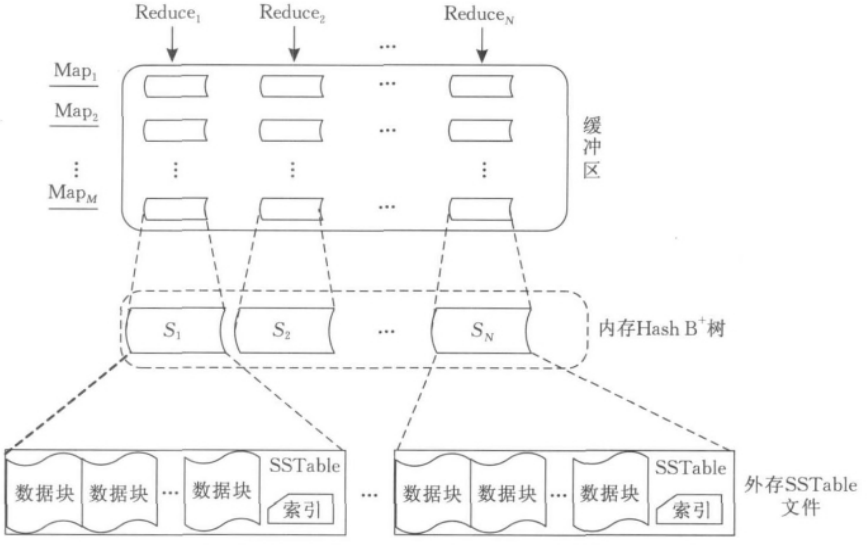

图 7 本地存储结构

3.3 外存读写策略

现有基于 SSTable 结构的文件读写策略是写优化的,如 BigTable[9] 在将内存缓存数据写(dump)到磁盘时采用直接写入一个新文件的追加写(minor compaction)方式,而在读时需要将缓存数据和若干个小文件进行合并(merge compaction),开销较大. 对于中间结果本地存储文件来说,读写操作都比较频繁并且比例均衡,不能盲目地只优化写操作. 为提高并发读写性能,可以根据开销选择读写方式. 下面给出各种读写开销的估算方法.

若寻道时间为常数 C_s,数据读写开销函数为 C_r 和 C_w,数据合并开销函数为 C_m.

追加写(dump by minor compaction)数据 d' 包括寻道和写数据开销,即

$$Cost_{dmic}=C_s+C_w(d').$$

合并读(read by merge compaction)已有数据 d 和新增数据 d' 包括两次寻道、两次读数据以及数据合并开销,即

$$Cost_{rmc}=2C_s+C_r(d)+C_r(d')+C_m(d,d').$$

合并写(dump by merge compaction)数据 d 包

括寻道和写数据开销，即

$$Cost_{\mathrm{dmec}}=C_{\mathrm{s}}+C_{\mathrm{w}}(d).$$

随机读(read randomly)数据 d 包括寻道和读数据开销，即

$$Cost_{\mathrm{r}}=C_{\mathrm{s}}+C_{\mathrm{r}}(d).$$

基于上述计算方法，在出现内外存替换时，对于要替换的 Hash 表项，应该首先利用 Map 和 Reduce 阶段间的缓冲区查看该表项是否即将被访问. 若此表项不会很快被访问，采用写开销较小的追加写方式；若此表项很快被访问，则比较

$$Cost_1=Cost_{\mathrm{dmic}}+Cost_{\mathrm{rmc}},$$

$$Cost_2=Cost_{\mathrm{dmec}}+Cost_{\mathrm{r}}.$$

若 $Cost_1$ 开销大，选择合并写和随机读方式，若 $Cost_2$ 开销大，选择追加写和合并读方式.

此外，针对追加写方式产生的小文件，通过管理线程进行合并以优化读操作，尤其在系统由于负载低而 CPU 空闲时，可以增加用于合并文件的管理线程.

3.4　替换算法

提高替换算法的命中率也是降低外存读写开销的重要方法. 在传统的操作系统页面替换算法中，最优算法[11]根据将要访问的页信息确定替换页，所以命中率最高，但在实际系统中，由于无法预知将要访问的页，该方法较少使用. 在 RTMR 的本地阶段化流水线中，Map 和 Reduce 阶段之间的缓冲区包含着将要访问的 Hash 表项信息，所以可以利用最优算法的思想. 此外，数据访问的局部性和替换代价也是需要考虑的因素. 因此，RTMR 内外存替换算法按照是否即将访问、是否最近访问、替换代价最小的顺序确定被替换的 Hash 表项. 其中，检索缓冲区数据确定表项是否将被访问，基于 LRU 算法记录表项的最近访问时间，按照数据量选择能容纳替入表项的最小表项.

4　基于系统参数的阶段化处理优化

为了提高数据流处理能力，RTMR 在每个工作节点构造阶段化流水线(如图 8)，利用线程池减少每次处理时的初始化开销，并通过异步的传递数据消除 Map 和 Reduce 阶段间的同步. 在阶段划分时，为减少阶段化的开销，将数据接收阶段和 Map 阶段合并，每个阶段由工作线程池、输入缓冲区和阶段内控制器组成，阶段之间通过控制器调整资源分配.

在 RTMR 本地阶段化流水线中，Map 和 Reduce 阶段各占用一部分工作线程，关键的共享资源是 CPU. 如何充分、有效地利用 CPU(包括 CPU Cache)是提高数据流处理能力的关键问题. 阶段化流水线可以通过阶段内的批调整和阶段间的线程池调整优化对 CPU 的利用.

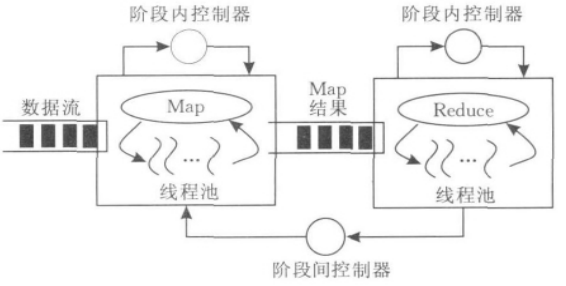

图 8　本地阶段化流水线

4.1　阶段内控制

批控制能提高阶段处理能力的原因在于：工作线程每次从缓冲区取一批数据的时间相当于取一条数据的时间，减少了访问缓冲区的次数；根据局部性原理，工作线程对一批数据的处理过程共享一个代码段和数据结构，可以有效提高 CPU Cache 的命中率，从而提高阶段处理速度. 在 SEDA 设计中，由于无法精确刻画批大小同阶段处理能力的关系，所以批控制一般基于启发式的反馈控制方式，通过观察系统参数调整批大小. 现有的 SEDA 批控制方法通过观察阶段吞吐量调整批，当吞吐量降低时增大批，当吞吐量增大时减小批. 为了说明这种方法的不足，给出如下定理.

定理 3.　阶段吞吐量不能决定处理速度.

证明.　设某阶段在时间 T 内处理了 n 条数据，吞吐量观察周期为 W，则阶段处理速度为

$$\mu=n/T,$$

吞吐量为

$$\eta=n/W.$$

根据排队稳态理论，当处理速度低于数据到达速度时，系统吞吐量由处理速度决定，此时吞吐量 η 提高(降低)说明处理速度 μ 提高(降低). 当处理速度不低于数据到达速度时，系统吞吐量由数据到达速度决定. 此时吞吐量的变化与数据处理速度无关.

证毕.

定理 3 说明以阶段吞吐量作为参数控制批大小会造成控制不准确，从而制约阶段处理能力提升. 为了充分利用阶段内的优化机制(主要是 CPU Cache)，应该以阶段处理速度作为控制参数，在没有达到最大处理能力之前，尤其是在数据处理速度跟上数据到达速度后，仍然可以增大批. 当处理速度开

始下降时，则应该减小批. 阶段内批调整算法如下.

算法 1. 批控制算法.

输入：数据到达速度 λ

输出：阶段处理速度 μ

1. $\lambda=0$, $\mu=0$, $\beta=1$;
2. 每批处理结束后，计算数据到达速度 λ'、阶段处理速度 μ'，已接收数据 d，工作线程数 n;
3. if $\mu'<\mu$
   $\beta=0.95\beta$;
4. else if $\mu'>\mu \wedge \mu<\lambda$
   $\beta=d/n$;
5. else if $\mu'>\mu \wedge \mu>\lambda$
   $\beta=1.05\beta$;
6. $\lambda=\lambda'$, $\mu=\mu'$;
7. 转步 2;

在算法 1 中，首先初始化数据到达速度、阶段处理速度和批大小. 之后在每批处理结束后，观察数据到达速度、阶段处理速度，若处理速度开始下降，则减小批；若处理速度增加但仍小于数据到达速度，则根据已接收数据增大批；若处理速度增加且已经跟上数据接收速率，则根据调整因子增大批，批的调整因子设为 5%.

4.2 阶段间控制

在阶段控制的基础上，阶段间控制器通过调整各阶段的工作线程以充分利用 CPU、提高整体吞吐量. 阶段间线程资源调整同样采用反馈控制方式，现有的 SEDA 方法通过观察阶段输入缓冲区内的数据规模调整线程池大小，当缓冲区数据超过阈值时增加线程，反之，减少线程. 这种方法没有综合考虑系统 CPU 使用信息进行各阶段间的全局控制，造成 CPU 利用率的提高无法转化为全局性能优化.

根据全局的 CPU 信息，造成 CPU 空闲（利用率低）的原因一是系统负载低，二是频繁发生读写文件等阻塞操作. 区别这两种情况的方法是判断数据处理速度是否跟上数据到达速度，若处理速度不低于数据到达速度，则属于前者. 反之，属于后者. 阶段间的工作线程调整主要针对第二种情况. 下面首先给出阶段过载的定义.

定义 5. 若阶段的数据处理速度低于数据到达速度，则称此阶段过载.

在阶段过载时，若 CPU 利用率不足，应该通过增加线程来提高 CPU 利用率，从而提高阶段处理速度. 然而，当 CPU 利用率到达一定程度后，如果继续增加线程，会因为频繁的线程切换导致系统开销增加，处理速度降低. 文献[7]指出，CPU 利用率在 75%以下时线程的切换开销成线性增长，而在 75%以上时则成指数增长. 因此，为了减少多线程的系统开销，将 75%作为判断是否可以增加线程的标准. 在一个阶段过载而另一个阶段非过载时，若 CPU 利用率小于 75%，增加过载阶段的线程数，若 CPU 利用率大于 75%，则移动非过载阶段的线程到过载阶段. 如果两个阶段都过载，由于系统吞吐量是由 Reduce 阶段决定的，所以应该优先向 Reduce 阶段增加或移动线程. 在确定了调整方法后，还应该确定调整目标，即系统何时达到吞吐量最大，下面定义了平衡系统的概念.

定义 6. Map 和 Reduce 阶段的数据到达速度分别为 λ_M, λ_R，处理速度分别为 μ_M, μ_R，CPU 利用率为 u. 若系统存在下列情况之一，则称系统是平衡的.

(1) $\lambda_R \leq \mu_R$, $\lambda_M \leq \mu_M$.

(2) $u \geq 0.75$, $\lambda_M > \mu_M$, $\lambda_R = \mu_R$.

定理 4. 平衡系统的吞吐量最大.

证明. RTMR 阶段化流水线可以看作一个由数据流接收、Map 和 Reduce 三个组件组成的串联系统，系统吞吐量由最小输出速度的组件决定.

若 $\lambda_R \leq \mu_R$ 并且 $\lambda_M \leq \mu_M$，即 Map 阶段和 Reduce 阶段的处理速度都大于数据到达速度，此时的系统吞吐量由数据流速度决定.

当 CPU 不低于 75%时，若 $\lambda_M > \mu_M$ 并且 $\lambda_R = \mu_R$，即 Map 阶段的数据处理速度低于到达速度，输出速度（Reduce 阶段的数据到达速度）与 Reduce 阶段的处理速度相等. 此时由于 CPU 繁忙，阶段间控制不能再增加线程，只能通过移动线程进行调整. 若移动 Map 阶段的工作线程到 Reduce 阶段，由于 Map 阶段的处理速度降低，造成 λ_R 降低，系统的吞吐量也就随之降低；若移动 Reduce 阶段的工作线程到 Map 阶段，虽然 λ_R 增加，但由于 Reduce 阶段处理能力下降造成系统吞吐量降低. 实际上，此时的系统处于过载平衡状态. 证毕.

定理 4 表明阶段间线程调整的目标是达到平衡状态. 阶段间工作线程调整算法如下.

算法 2. 线程池调整算法.

输入：Map 和 Reduce 阶段的数据到达速度 λ_M, λ_R，CPU 利用率 u

输出：Map 和 Reduce 阶段的处理速度 μ_M, μ_R

1. 每 5 秒观察 Map 和 Reduce 阶段的数据到达速度 λ_M, λ_R，处理速度 μ_M, μ_R 以及全局 CPU 利用率 u;
2. if $\lambda_R=\mu_R \wedge u \geq 0.75$
   转步 1;
3. else if $\lambda_R \leq \mu_R \wedge \lambda_M \leq \mu_M$

转步 1；

4. else if $\lambda_R > \mu_R$
　　if $u < 0.75$
　　p_R++；
　　else
　　p_R++；p_M−−；

5. else if $\lambda_M > \mu_M$
　　if $u < 0.75$
　　p_M++；
　　else
　　p_M++；p_R−−；

6. 转步 1.

在算法 2 中，首先向过载阶段增加或移动工作线程，当两个阶段都过载时，优先考虑向 Reduce 阶段增加或移动工作线程，直至达到平衡状态. 算法的调整周期设定为 5 s，这样能够保证阶段内的批控制方法能够有充足的时间适应线程池的变化.

在系统达到平衡状态后，若系统非过载且 CPU 利用率小于 75%，说明 CPU 没有被充分利用就已经能够跟上数据流速度，则线程太多只会增加开销；若 CPU 利用率高于 75%，则会由于线程太多造成开销急剧增加；在系统过载情况下，CPU 利用率肯定大于 75%，此时也会由于线程太多导致开销增加. 因此，在系统达到平衡状态后，无论系统是否过载，都可以试图通过减少线程降低系统开销. 具体的调整方法为依次减少 Map 和 Reduce 阶段的线程直到阶段处理速度开始降低或重新达到不平衡状态.

5　评　价

本节以物联网环境下城市车辆实时数据处理应用中具有代表性的套牌车计算作为基准测试验证 RTMR 方法.

5.1　基准测试

套牌车主要根据车辆时空矛盾来判定，针对每一条出现在特定监控地点的车辆实时数据，检索该车牌出现在所有其他监控点且在最大套牌时间阈值内的历史数据，如果二者的时间差小于该两点之间的套牌时间阈值，则认为该车辆有套牌嫌疑. 根据前期物联网建设工作统计数据，若在某大型城市全面捕获车辆数据，数据流速度将会达到 1 MB/s（每条数据按 200 B 计，约 5000 条/s），同时，在车辆数据保存一个月的情况下，历史数据的存储量将达到 1 TB.

套牌车计算可以使用如下 RTMR 算法：针对某车牌号，Map 阶段在所有车牌号的 Hash 表中找到其所在分组表项，Reduce 阶段在 B^+ 树中找到其链表所在位置，依次与每条历史数据比较套牌时间阈值并更新链表. 在某城市的车牌数量达到 10^7 的情况下，可以采用输出为 20 位二进制数的 Hash 函数

$$\mathrm{Hash}(k) = k \bmod 2^{20}$$

进行分组，存储中间结果的 Hash 表共有 2^{20} 个表项，平均每个 Hash 表项存储 $10^7/2^{20} \approx 10$ 个车牌的数据. 因为在持久化历史数据中可能只有部分数据与计算相关，例如在套牌车计算中只有套牌阈值范围内的历史数据相关，所以以预处理的历史数据中间结果规模来衡量流处理架构处理能力.

数据流处理集群搭建在 2×6 核 2.0 GHz CPU、32 GB 内存、250 GB 硬盘的服务器集群上；使用 Oracle 10g 作为持久化存储，搭建在一台 2×4 核 2.4 GHz CPU，16 GB 内存服务器和 20 TB RAID 5 磁盘阵列上；网络连接采用 1 Gbps 以太网光纤和交换机；在 1 台双核 3.0 GHz CPU、4 GB 内存服务器上使用 Load Runner 9.0 模拟数据流. 为了测试数据流处理架构的伸缩能力，在集群节点上均匀划分历史数据区间，并在车辆数据流随机性和局部性特点基础上，模拟在集群节点上均匀分布的数据流，具体方法为：以十进制区间 $(0, 10^7]$ 内的数模拟车牌号，若存在 n 个节点，在各节点的历史数据划分区间中选取一个子集 $P'_1, P'_2, \cdots, P'_n$，使得 $|P'_1| + |P'_2| + \cdots + |P'_n| = 10^5$，循环的为 n 个节点产生负载，对于节点 i，在 P'_i 中选取一个随机表项 t，在区间 $(0, 10)$ 内选取一个随机数 x，以 $2^{20}x + t$ 作为该条模拟数据的车牌号，随机设定监控点，记录系统时间添加时间戳并控制数据流速度.

本文在测试单个方法优化效果时，采用面向方面编程（Aspect-Oriented Programming，AOP）思想，禁用其他优化方法，将测试代码插入测试目标方法，作为比较对象的方法使用同样的集群配置同时接收和处理数据流负载，分别计算性能指标.

5.2　中间结果存储性能分析

首先验证 RTMR 对中间结果本地存储的优化效果. 实验使用单个节点，数据流速度固定为 1 MB/s（每条数据 200 B，约 5000 条/s），中间结果数据规模为 50 GB，每项测试进行 10 次，每次 10 min，取平均值计算实验结果. 表 1 显示了对中间结果本地存储进行优化前后的性能对比，可以看到，RTMR 通过建立缓冲区消除读写同步，使中间结果的内存读写

性能提高了12.1%,RTMR通过利用缓冲区信息指导读写策略和替换算法,使外存读写性能和内外存命中率分别提高了15.5%和9.3%.综合3种方法,将中间结果本地读写性能(单位时间内读写次数)提升了近1/4.

表1 中间结果存储性能优化

性能指标	测试方法	实验结果	效果/%
内存读写性能	读写同步	75385.2	12.1
	同步消除	84506.8次/s	
外存读写性能	BigTable	4425.5次/s	15.5
	RTMR	5111.4次/s	
内存命中率	LRU	66.7%	9.3
	RTMR	72.9%	
整体读写性能	改进前	73901.4次/s	24.8
	RTMR	88608.5次/s	

在上述实验的基础上,保持数据规模不变,逐步提高数据流速度,测试RTMR方法外存读写性能和命中率的变化情况.由图9和图10可知,随着数据流速度的提高,由于缓冲区队列不断变大,能够为外存读写策略和替换算法提供更为充足的将要访问表项信息,因而读写性能和命中率不断提高.但当数据流速度提高到一定程度时,缓冲区数据规模的扩大将不会再引起读写性能和命中率的提高,反而因为检索开销加大,以及占用过多内存资源引发了额外的内外替换和读写操作,造成读写性能和命中率急剧下降.针对上述规律,在RTMR内外存结构、读写策略和替换算法的基础上,下一步的工作是对数据流速度、缓冲队列大小与读写性能、命中率的关系进行进一步分析,以指导缓冲区大小的动态变化以及负载丢弃策略.

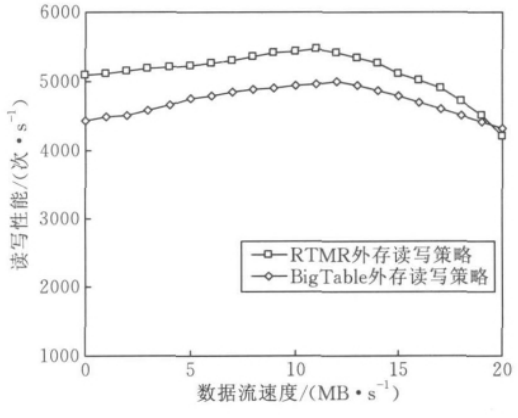

图9 RTMR读写策略分析

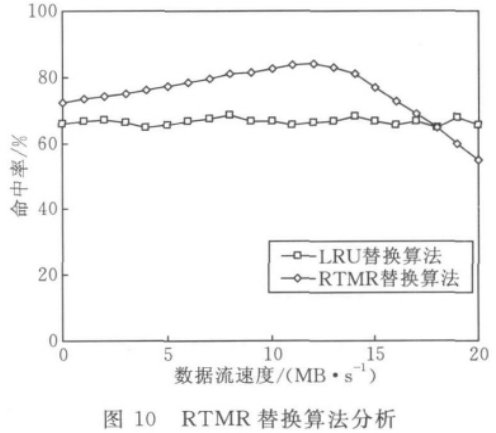

图10 RTMR替换算法分析

5.3 阶段化流水线性能分析

与5.2节实验类似,验证阶段化流水线优化效果的实验使用单个节点,数据流速度固定为1 MB/s,中间结果数据规模为50 GB,每项测试进行10次,每次10min,取平均值计算实验结果.表2显示了对本地阶段化流水线进行优化前后的性能对比.可以看到,由于RTMR利用阶段数据到达速度和处理速度进行控制,比利用吞吐量进行控制更能充分利用CPU Cache,因此Reduce阶段处理速度提高了14.8%;由于RTMR根据CPU信息进行各阶段间的全局控制,更能将CPU利用率充分转化为性能提升,因此系统吞吐量提高了10.7%.综合两种方法,将阶段化流水线的整体性能(吞吐量)提升了近1/5.

表2 阶段化流水线性能优化

性能指标	测试方法	实验结果/(条·s⁻¹)	效果/%
阶段处理速度	SEDA	7848.8	14.8
	RTMR	9010.4	
吞吐量	SEDA	4804.8	10.7
	RTMR	5319.0	
流水线性能	SEDA	4920.7	19.2
	RTMR	5746.3	

在上述实验基础上,保持数据规模不变,分别以Reduce阶段处理速度和系统吞吐量为测试目标,采用变化的数据流比较RTMR与SEDA的批控制和线程池控制效果.实验进行10次,记录60 s内阶段处理速度和系统吞吐量的变化,取平均值计算实验结果.SEDA控制方法的配置取实验最优值,RTMR线程池控制周期配置为与SEDA相同的500 ms.图10和图11中黑线所示为实际数据流负载对应的阶段处理速度和吞吐量参考值.

由图11可知,SEDA批控制方法在吞吐量降低的情况下增大批(阶段处理速度提高),在吞吐量增大的情况下减小批(阶段处理速度降低),造成在数据流速度提高的情况下需要较长的调整时间,而在数据流速度降低情况下衰减又太快,并且由于未能

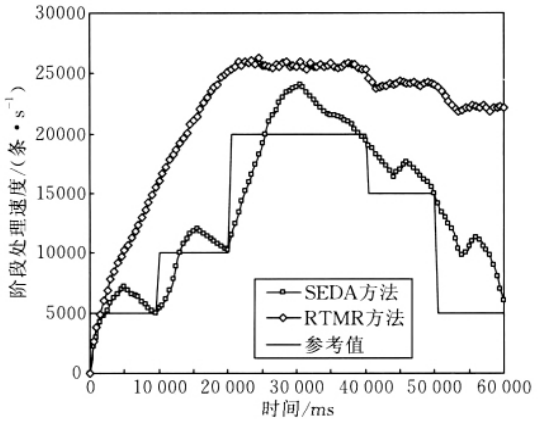

图 11　批控制分析

充分利用 CPU Cache 等阶段优化机制，制约了阶段处理速度. 而 RTMR 方法通过观察阶段数据到达速度和处理速度持续增大批，能够在短时间内达到更高的阶段处理速度并在数据流速度降低的情况下缓慢衰减. 由图 12 可知，SEDA 方法根据缓冲数据规模变化调整线程池，需要较多次调整才能适应数据流，并且由于没有考虑整个系统的瓶颈，单个阶段线程池的调整无法转化为全局吞吐量提升，而 RTMR 方法根据 CPU 信息进行全局调整，使得 CPU 得到了充分有效利用，因而在调整稳定后吞吐量更高.

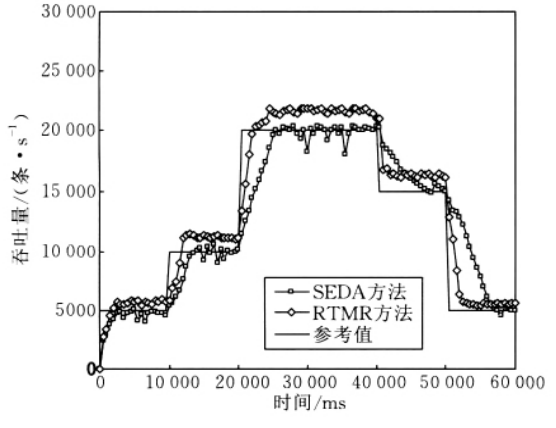

图 12　线程池控制分析

5.4　实时性分析

在实时性分析实验中对比 S4、HOP 和 RTMR 3 种处理架构. 由于 S4、HOP 不支持预处理，每次处理数据流时都要重新进行历史数据加载和处理，大量无谓开销制约了系统吞吐量. 因此，为了比较面向大规模数据的数据流处理能力，在基准测试的 S4 和 HOP 实现中加入了预处理逻辑，也就是先将相关历史数据进行一次流处理. 各种数据流处理架构都搭建在两个节点上，数据流速度固定为 1 MB/s，每种数据规模测试 10 次，每次 10 min，取平均值计算实验结果.

由图 13 和图 14 可知，当中间结果规模小于 32 GB 时，因为一个节点的内存就可以容纳全部中间结果，所以 HOP 和 S4 也具有很高的吞吐量（大于 4500 条/s），但 RTMR 由于采用了本地阶段化流水线和内存读写优化，吞吐量更高；当中间结果规模超过 32 GB 时，中间结果分布到两个节点的内存中，HOP 和 S4 由于节点间的数据传输和同步开销增加造成吞吐量下降较快，而 RTMR 由于采用本地化技术避免了数据传输，吞吐量依然很高；当中间结果规模超过 64 GB 时，由于已经超出了 S4 和 HOP 的中间结果缓存能力，其吞吐量趋于稳定，但错误率随数据规模增加而变大，而 RTMR 由于利用本地外存对中间结果缓存能力进行了扩展和优化，能够在降低错误率的同时（低于 5%）保持较高的吞吐量（大于 4300 条/s）.

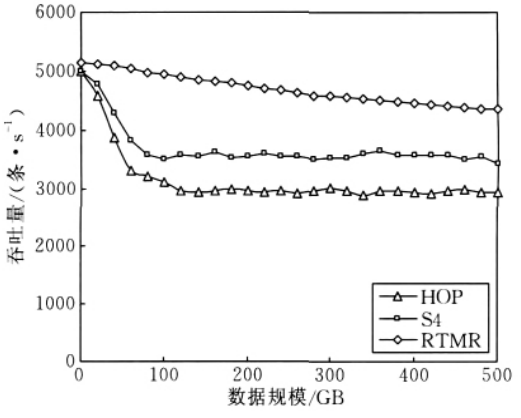

图 13　性能对比

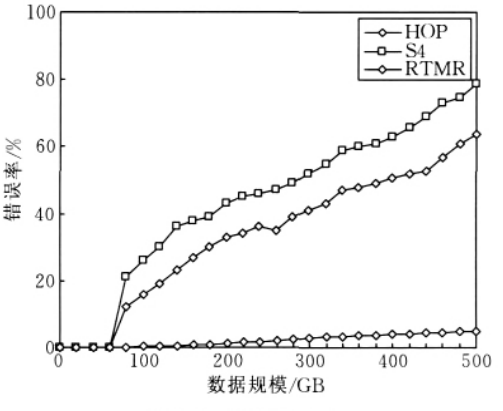

图 14　错误率对比

5.5　可伸缩性分析

在伸缩性分析中进行两组实验，分别测试 RTMR 针对历史数据规模和数据流速度的伸缩能力.

在第 1 组实验中固定数据流速度为 1 MB/s，测试在节点数量增加情况下 RTMR 所能处理的历史

数据规模.由图 15 左 y 轴曲线可知,节点增加时 RTMR 处理能力的提升是近似线性的,这是由于 RTMR 通过划分历史数据中间结果和本地化处理,使节点之间不会产生制约并行吞吐量提升的数据传输和同步开销.而之所以未能达到线性伸缩,是因为在历史数据规模扩大的情况下本地存储文件读写开销增加.

在第 2 组实验中固定每个节点中间结果数据规模为 50 GB,测试在节点数量增加情况下 RTMR 所能处理的数据流.由图 15 右 y 轴曲线可知,在数据流速度低于 15 MB/s 的情况下,节点增加时 RTMR 处理能力的提升是近似线性的,在数据流速度超过 15 MB/s 的情况下,节点增加时 RTMR 处理能力的提升变缓.这是因为随着数据流速度的提高,RTMR 每个节点接收冗余数据流和进行 Map 阶段的 CPU 开销增加,同时 RTMR 方法的中间结果读写性能和替换命中率开始下降,所以影响了吞吐量.

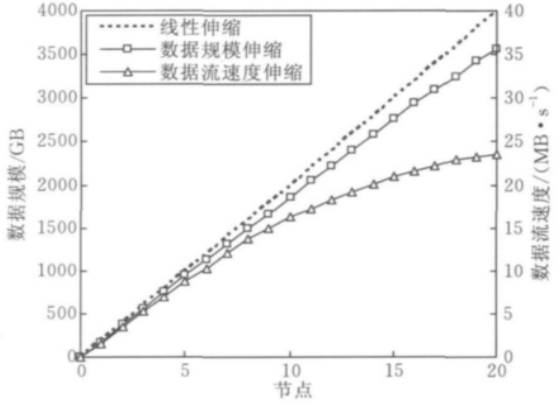

图 15 伸缩性分析

从利用 RTMR 方法解决车辆监管问题的经验来看,现阶段物联网环境下的数据流处理应用,受采集端带宽等因素影响,数据流速度远达不到 15 MB/s,只需占用节点多核 CPU 的很小一部分就可以完成接收和分组操作,并且 RTMR 方法能够有效提高中间结果的并发读写性能,在历史数据规模不断扩大的情况下,RTMR 具有很好的伸缩性.

6 相关工作

编程模型是连接特定问题与特定硬件实现的桥梁,并行编程模型更是建立大规模数据处理环境的重要基础.典型的并行编程模型有 OpenMP、MPI、MapReduce 和 Dryad[12] 等,其中 OpenMP 和 MPI 是抽象层次比较低的模型,需要程序员显式处理任务管理和数据管理等细节;Dryad 侧重于构造一个完整的计算流程;而 MapReduce 则侧重于构造一个计算算子.对于可划分的大规模数据处理任务,MapReduce 能够提供充分的并行计算语义,因而基于批处理方式的 MapReduce 得到了广泛应用,已有在多核系统、多核集群系统上的多种实现.然而,以批处理方式处理持续到达的数据流,无法满足实时需求.

针对 MapReduce 的实时性改进已经成为研究热点,增量处理 Percolator[13]、DryadInc[14] 和迭代处理 Twister[15]、Spark[16] 等工作通过随机访问存储、缓存和复用中间结果提高了大规模数据处理的实时性.然而,上述方法仍属对静态数据增量的批处理方式.HOP[17] 和 S4[18] 分别利用流水线和分布处理单元技术扩展了 MapReduce 的实时处理能力,但 S4 和 HOP 依然不是以面向大规模历史数据的数据流处理为目标的,表现在编程模型中缺乏对历史数据预处理和同步方法的支持,也没有充分利用中间结果缓存和本地阶段化处理等技术优化 MapReduce 的数据流处理能力.本文基于 MapReduce 模型提出了一种支持此类数据处理的方法 RTMR,并对其中的关键技术瓶颈进行了改进.

虽然 S4 和 HOP 都采用了流水线技术,但 S4 需要在节点间传输大量中间结果,HOP 需要在 Reduce 阶段进行前完成多个 Map 节点结果的同步合并,都没有充分利用多核计算资源本地化 MapReduce 处理,实时处理能力受限于节点之间的数据传输和同步.Phoenix[6] 和 Metis[8] 实现了多核架构上的 MapReduce,但仍属于批处理方式.文献[7]提出的阶段化事件驱动架构 SEDA 能够利用线程池技术减少每次处理的初始化开销,并通过在阶段间异步传递数据消除阶段间的同步;还可以通过控制阶段内每次处理的批大小和阶段间的资源调整提高实时处理能力.本文基于 SEDA 构建了 RTMR 本地阶段化流水线.然而,现有的 SEDA 控制器[7] 依赖于经验或实验方式,没有充分利用各种系统参数进行自动控制,往往花废大量时间仍不能达到最优的控制效果,造成 CPU(包括 CPU Cache)无法得到有效利用,从而制约了数据流处理能力.系统控制方法可以分为准入控制和反馈控制两种,准入控制方法简单,但需要较多的人工参与并且控制策略固定、调整速度慢.现有的 SEDA 线程池控制方法就属于此类.反馈控制的优势在于自动调整能力,文献[19]通过 PI 控制适应性的调整线程池,以提高资源利用率、优化性能,文献[20]基于自动 agent,通过附加调整参数对 MIMO 系统进行优化控制.本文以反馈控制方式改造了 SEDA 控制方法,与上述

工作相比，通过分析系统输入输出参数与控制参数的关系，确定控制目标和控制方式，使得对批大小和线程池的控制更为准确. 此外，本工作在线程池控制中利用 CPU 使用信息进行各个阶段的全局控制，使 CPU 得到了充分有效利用，提高了数据流处理能力.

数据流处理系统 S4，迭代系统 Twister、Spark 和增量处理系统 DryadInc 将中间结果以对象或键值形式缓存在本地内存中，通过复用提高实时处理能力. 然而，上述工作都没有着重解决中间结果内存结构的高并发读写性能问题. Metis 采用 Hash B^+ 树结构存储中间结果. 与对象、Hash 表和树等结构相比，Hash B^+ 树针对各种负载类型都有很高的读写性能. 本文在 Hash B^+ 树基础上，通过消除阶段化流水线中 Map 和 Reduce 线程间的读写同步进一步优化中间结果的并发读写性能. 此外，上述工作的中间结果规模受制于内存容量，没有充分利用本地存储（包括外存）的数据存储能力，文献[21]在外存上建立了历史数据中间结果存储，但由于采用抽样方法丢失了部分历史数据，所以主要用于针对历史数据的聚集查询，不支持数据流同历史数据实时计算所需的随机查询. 本文基于 SSTable 结构建立了键值类型历史数据中间结果的外存文件. 然而，现有基于 SSTable 的文件读写策略[9]只是对写操作进行优化，读开销较大，应对频繁发生并且比例均衡的中间结果读取和写入操作，具有一定盲目性. 另外，从提高内外存替换命中率角度，现有的 LRU、clock 等替换算法[11]没有充分利用阶段化流水线缓冲区中包含的将要访问的表项信息，制约了替换命中率. 本文利用读写开销估算和缓冲区信息改造外存文件读写策略和内外存替换算法，进一步优化了中间结果的高并发读写性能.

7　结束语

物联网环境下针对高速数据流的大规模数据处理难点在于伸缩性和实时性两方面的需求，本文提出了一种支持此类数据处理的 RTMR 方法，其创新性主要表现在：

（1）通过中间结果分布缓存和本地阶段化流水线技术，改进了 MapReudce 的数据流实时处理能力.

（2）根据系统参数控制本地阶段化流水线，使 CPU 得到了充分、有效利用，提高了实时处理能力.

（3）通过改造内外存数据结构、读写策略和替换算法，优化了本地中间结果的高并发读写性能.

基于实际物联网环境的基准测试表明 RTMR 方法能够保障大规模历史数据上数据流处理的实时性和可伸缩性，具有很高的应用价值. 在 RTMR 方法中，单个节点处理能力的提升是集群可伸缩性的基础，因此本文主要解决本地优化问题. 下一步工作主要针对 RTMR 可伸缩性保障中的关键问题——负载均衡，包括异构工作节点上的中间结果数据分布和动态负载调度.

参　考　文　献

[1] Motwani R, Widom J, Arasu A et al. Query processing, resource management, and approximation in a data stream management system//Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR 2003). Asilomar, USA, 2003: 176-187

[2] Abadi D J, Ahmad Y, Balazinska M et al. The design of the Borealis stream processing engine//Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR 2005). Asilomar, USA, 2005: 277-289

[3] Chandrasekaran S, Cooper O, Deshpande A et al. TelegraphCQ: Continuous dataflow processing for an uncertain world//Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR 2003). Asilomar, USA, 2003: 200-211

[4] Gibbons P, Matias Y. Synopsis data structures for massive data sets//Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1999). Baltimore, USA, 1999: 909-910

[5] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. ACM Communication, 2008, 51(1): 107-113

[6] Ranger C, Raghuraman R, Penmetsa A, Bradski G, Kozyrakis C. Evaluating MapReduce for multi-core and multiprocessor systems//Proceedings of the 13th International Conference on High-Performance Computer Architecture (HPCA 2007). Phoenix, USA, 2007:13-24

[7] Welsh M, Culler D, Eric Brewer E. SEDA: An architecture for well-conditioned, scalable Internet services//Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP 2001). Lake Louise, Banff, Canada, 2001: 230-243

[8] Kaashoek F, Morris R, Mao Y. Optimizing MapReduce for multicore architectures. MIT Computer Science and Artificial Intelligence Laboratory: Technical Report MIT-CSAIL-TR-2010-020, 2010

[9] Chang F, Dean J, Ghemawat S et al. Bigtable: A distributed storage system for structured data//Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006). Seattle, USA, 2006: 205-218

[10] DeCandia G, Hastorun D, Jampani M et al. Dynamo: Amazon's highly available key-value store//Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007). Stevenson, USA, 2007: 205-220

[11] Lubomir F B, Show A C. Operation System Principles. New Jersey: Prentice Hall, 2003

[12] Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: Distributed data-parallel programs from sequential building blocks//Proceedings of the 2007 Eurosys Conference (Eurosys 2007). Lisbon, Portugal, 2007: 59-72

[13] Peng D, Dabek F. Large-scale Incremental processing using distributed transactions and notifications//Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010). Vancouver, Canada, 2010: 251-264

[14] Popa L, Budiu M, Yu Y, Isard M. DryadInc: Reusing work in large-scale computations//Proceedings of the 2009 Conference on Hot Topics in Cloud Computing (HotCloud 2009). San Diego, USA, 2009: Article 21

[15] Ekanayake J, Li H, Zhang B, Gunarathne T, Bae S, Qiu J, Fox G. Twister: A runtime for iterative MapReduce//Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC 2010). Chicago, IL, USA, 2010: 810-818

[16] Zaharia M, Chowdhury N M, Franklin M, Shenker S, Stoica I. Spark: Cluster computing with working sets//Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud 2010). Boston, USA, 2010: 1-10

[17] Condie T, Conway N, Alvaro P, Hellerstein J M, Elmeleegy K, Sears R. MapReduce online//Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2010). San Jose, USA, 2010: 313-328

[18] Neumeyer L, Robbins L, Nair A, Kesari A. S4: Distributed stream computing platform//Proceedings of the 10th IEEE International Conference on Data Mining Workshops (ICDMW 2010). Sydney, Australia, 2010: 170-177

[19] Lu C, Abdelzaher T F et al. A feedback control architecture and design methodology for service delay guarantees in web servers. Department of Computer Science, University of Virginia: Technical Report CS-2001-06, 2001

[20] Diao Y, Hellerstein J L et al. Managing web server performance with autotune agents. IBM System Journal, 2003, 42(1): 136-149

[21] Shah M A, Hellerstein J M, Chandrasekaran S et al. Flux: An adaptive partitioning operator for continuous query systems//Proceedings of the 19th International Conference on Data Engineering (ICDE 2003). Bangalore, India, 2003: 25-36

QI Kai-Yuan, born in 1984, Ph. D. candidate. His current research interests include large scale data processing and service computing.

ZHAO Zhuo-Feng, born in 1977, Ph. D., associate researcher. His current research interests include service computing, large scale data processing and cloud computing.

FANG Jun, born in 1976, Ph. D., assistant researcher. His current research interests include service computing and cloud computing.

MA Qiang, born in 1986, M. S. candidate. His current research interests include large scale data processing and data stream processing.

Background

Large scale data processing and data stream processing are classical research topics in data management. With the development of cloud computing and Internet of Things, computing based on real-time and historical sensor data becomes the key point to the Internet of Things applications, and how to solve the real-time computing for high speed data stream over large scale persistent data brings a new challenge. The existing batch processing based MapReduce large scale data processing architecture cannot satisfy the real-time requirement, and the previous data stream processing architecture cannot deal with large scale historical data. This paper first proves that the MapReduce model can be used for large scale data processing under high speed data stream, then proposes a new MapReduce architecture for such kind of applications, and removes some technical bottlenecks such as local staged pipeline and intermediate result storage. Based on a benchmark from a real IoT application, we can see this MapReduce architecture improves performance and scalability for data stream processing over large scale data compared with previous work. This work was supported by the National Natural Science Foundation of China under Grant Nos. 60903137 and 61003294.