文章编号：2095-2163(2024)03-0037-09 中图分类号：TP391;TP311 文献标志码：A

面向并发数据流处理的存储接入方法

胡汝博，陈庆奎
（上海理工大学 光电信息与计算机工程学院，上海 200093）

摘 要：海量的并发数据流存储请求，会造成数据库瞬时压力过大和存储时延提高等问题，对此提出一种面向并发数据流处理的存储接入方法。首先构建 Redis 缓冲区队列，以接收上游数据处理节点的周期数据，使用自定义数据流协议解决数据流规范化通信；然后设计主动存储中间件，通过线程读写任务分配算法（TRWTA），优先处理高负载区队列的读写任务，利用动态线程池计算读写压力（RWP）来优化管理线程数量，达到提高并发数据流存储效率的目的。此外，在存储层中使用了 Mysql 和 HBase 对数据流进行分类存储，对 HBase 进行 Rowkey 设计和预分区。经过实验分析，所提存储接入方法在持续的并发数据流下，较传统存储方法有效降低了存储时延，保证了数据库稳定性。

关键词：并发数据流；主动存储中间件；读写任务分配；动态线程池；存储接入

Storage access method for concurrent data stream processing

HU Rubo, CHEN Qingkui
（School of Optical-Electrical and Computer Engineering, University of Shanghai for Science and Technology, Shanghai 200093, China）

Abstract: Massive concurrent data stream storage requests can cause excessive transient pressure on databases and increase storage latency. This paper proposes a storage access method for concurrent data stream processing. Firstly, a Redis buffer queue is constructed to receive periodic data from upstream data processing nodes, and a customized data flow protocol is used to solve data flow standardization communication. Then, an active storage middleware is designed to prioritize read and write tasks for queues in high load areas through a thread read and write task allocation algorithm (TRWTA). Dynamic thread pools are also used to calculate read and write pressure (RWP) to optimize the number of managed threads, achieving the purpose of improving the storage efficiency of concurrent data streams. In addition, MySQL and HBase are used in the storage layer to classify and store data streams, and HBase is designed for Rowkey and pre-partitioning. Through experimental analysis, the proposed storage access method effectively reduces storage latency and ensures database stability compared to traditional storage methods under continuous concurrent data streams.

Key words: concurrent data flow; active storage middleware; assignment of reading and writing tasks; dynamic thread pool; storage access

0 引 言

随着边缘计算[1-2]的飞速发展，采集设备数量呈爆炸性增长，导致海量数据信息需要进行实时计算，转化为数据处理流。终端设备采集的周期性，决定了数据处理的周期性，也影响了存储的周期性。数据流包含唯一标识、时间戳、数据区和处理周期，而并发数据流是指具有大量相同周期的数据流的集合[3-4]。传统存储方法将并发数据流直接写入数据库，这样会给数据库造成巨大的存储压力。因此，如何构建一种可以降低数据库存储压力的并发数据流存储接入方法，已成为迫切需要研究的方向。

Kolajo 等学者[5]为并发数据流的存储提供了思路，提出将资源分配并行化处理以应对日益增长的数据规模。涂聪等学者[3]通过多处理节点能力的不同进行分流，把庞大的数据流处理压力分摊到多个节点中。但数据节点过多会因数据库连接过多导

基金项目：国家自然科学基金（61572325）；上海重点科技攻关项目（16DZ1203603，19DZ1208903）；上海智能家居大规模物联共性技术工程中心项目（GCZX14014）；上海市一流学科建设项目（XTKX2012）；上海市重点项目（9DZ1208903）。

作者简介：胡汝博（1996-），男，硕士研究生，主要研究方向：并行计算、并发数据流处理、边缘集群存储架构研究，Email：rubo258@163.com；陈庆奎（1968-），男，博士，教授，博士生导师，CCF 会员，主要研究方向：计算机集群、人工智能、并行理论、物联网大规模数据分析等。

收稿日期：2023-03-13

致数据量持续提升,这给存储性能带来巨大的挑战。为了能够实现并发数据流数据高效存储,应该主动把控存储的速度来缓解存储压力。陆铭琛等学者[6]提出了使用多个数据桶构建单逻辑下的轮转存储模型,将数据桶分配给不同存储节点,有效提升了写入吞吐量,但实现难度较大。

此外,并发数据流可以考虑使用缓冲区降低存储压力,陆婷等学者[7]提出多源缓冲区,但未能考虑缓冲区负载变化对存储性能的影响,因此需要一种可以缓冲上游各数据处理节点的缓存区。为了减轻并发数据流的存储压力,传统的固定线程已不能满足需求,动态线程池[8-10]提供了可以及时回收和增加线程的机制来提高存储效率。赖德军等学者[11]根据任务量自动创建线程,并定期清除冗余线程,但没有考虑系统中多个资源对线程的约束。

近年来,在面对并发数据流数据库的选型上,对于数据量可控的常规结构化数据来讲,传统关系型数据库即可满足存储需求。但传统关系数据库处理海量数据,通常利用中间件进行分库分表、索引管理,但面临维护困难、数据检索延时高等问题,无法支撑海量数据的增长。为了应对这些挑战,日志索引结构树(Log-Structured Merge Tree,LSM-tree)[12]备受研究者关注,得益于解决了存储引擎 B 树(B-tree)写入开销大等问题,其中应用于其一的列式存储 HBase[13]越来越受欢迎。HBase 基于 HDFS 文件系统建立,在存储海量结构化、非结构数据时有较大的优势。例如:通过横向拓展,解决了存储瓶颈问题。此外,HBase 还在解决物联网存储方面发挥重要的作用,如 Wang 等学者[14]利用 HBase 实时读写特性,成功应用在海量的地理时空分类存储。HBase 存储关键是 Rowkey 的设计[15],通常遵循唯一性、散列性、长度尽可能短的原则,但设计不良易造成存储的数据倾斜。

为了解决并发数据流的上游数据传输与下游存储的问题,设计存储中间件具有重要意义。Zhang 等学者[16]指出了中间件技术未来所面临的挑战,如在上下文感知的主动适应欠佳、资源分散难以高效管理等。Pradeep 等学者[17]设计了基于多代理和多通信协议支持的上下文感知的自适应中间件,提出按照感知服务划分不同层次,再利用异构事务主动监控可用计算资源,支持上下文感知和实现最小的平均响应时间。传统存储架构[18-19]面对不断扩大的数据量,会因磁盘 I/O 压力过大、存储线程阻塞造成存储时延[20]提高问题。

综上研究,均未能提供面向并发数据流处理的存储接入方法,未能解决当上游数据处理节点过多导致数据库瞬时连接过多、存储压力过大的问题。本文研究了并发数据流动态处理与配置的存储问题,由此提出一种面向并发数据流处理的存储接入方法,将从系统架构及核心思想、系统实现和实验分析三部分进行研究,以此提高存储效率。

1 系统架构及核心思想

1.1 总体架构

本文背景为采集海量图片的并发数据流处理系统。如图 1 所示,该系统的总体架构由采集接入系统、数据处理集群、Redis 缓冲区、主动存储中间件、持久化存储等模块构成。在数据处理节点中,数据抓取服务从采集接入系统周期抓取,经环形缓冲区发送到数据计算服务。在处理周期中由数据抓取服务产生的链接资源流,包含了图片的链接描述等信息;数据计算服务输出图片处理后的处理结果流,包含了由图片计算后的各种指标值;服务监控输出服务监控流,包含了数据抓取服务的活跃度、数据计算能力分析等。上述 3 种数据流具有周期性,并按照数据处理的周期推送到 Redis 缓冲区,每个数据处理节点对应一个缓冲区,且包含 3 种队列分别缓冲上述 3 种数据流。

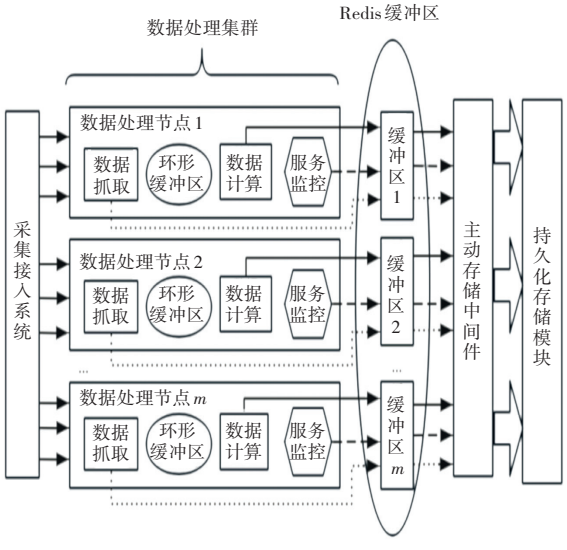

图 1 总体架构

Fig. 1 Overall architecture

1.2 存储接入架构

本文聚焦数据处理节点下游,面向并发数据流

处理的存储接入问题，提出的存储接入架构如图 2 所示。

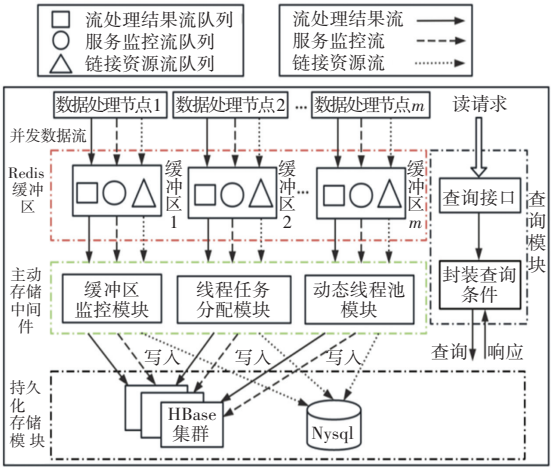

图 2　存储接入架构示意图

Fig. 2　Schematic diagram of storage access architecture

存储接入架构整体由 Redis 缓冲区、主动存储中间件、持久化存储模块、查询模块等 4 部分组成。

(1) Redis 缓冲区。Redis 缓冲区构建多缓冲区队列，其中包括数据流协议和缓冲区队列长度自适应调整等，用来缓冲不同数据流。每个缓冲区队列周期接收上游数据处理节点的流处理结果流、服务监控流和资源链接流。

(2) 主动存储中间件。主动存储中间件包括缓冲区监控模块、线程读写任务分配模块、动态线程池模块。其中，缓冲区监控模块定时收集 Redis 缓冲区队列监控表内容，并计算综合负载等信息同步到 Redis 缓冲区负载表；线程读写任务分配模块优先分配线程处理综合负载最高的缓冲区，通过周期收集缓冲区不同队列数据，批量存储在指定数据库中；动态线程池模块根据线程池读写压力(Read Write Pressure，RWP)，动态添加或减少线程，提高系统资源利用率。

(3) 持久化存储。持久化存储层利用关系数据库 MySQL 和列式数据库 HBase 进行存储。对于处理结果的流数据、服务监控流数据具有实时写入与查询需求存储在 HBase 集群中，并设计 Rowkey 以及预分区管理；而资源链接流因数据量可控存储在 MySQL 中。

(4) 查询模块。查询模块提供规范化的每个数据流查询接口，其中包括请求和调用方式等，方便了前端界面的调用。

2　系统实现

2.1　Redis 缓冲区实现

2.1.1　Redis 缓冲区数据传输

2.1.1.1　数据流协议

为了方便 Redis 缓冲区上下游的通信，数据流协议适配不同数据流的数据包进行传输。Redis 缓冲区数据包格式如图 3 所示。由图 3 可知，数据包格式包含头部、数据域和校验字。对此将做研究论述如下。

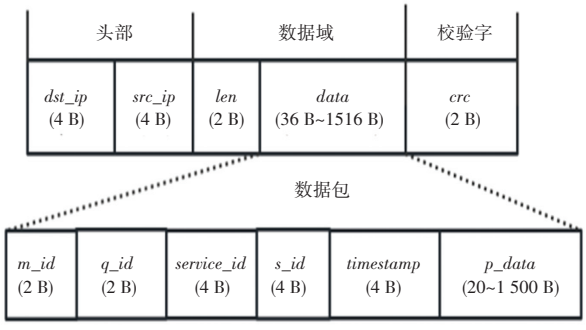

图 3　Redis 缓冲区数据包格式

Fig. 3　Redis buffer packet format

(1) 头部：*dst_ip* 为目标 IP；*src_ip* 为源 IP。

(2) 数据域：*len* 为报文的长度；*data* 为可变大小的数据区。*data* 中包括缓冲区 ID (*m_id*)、队列序号(*q_id*)，其中序号 1 为流处理结果流队列、序号 2 为服务监控流队列、序号 3 为链接资源流队列；*service_id* 为数据处理节点 ID；*timestamp* 为时间戳；*p_data* 为数据区。

(3) 校验字：*crc* 为数据域的循环冗余校验码，用于验证数据包完整性。

协议封装以下 3 种流数据包，通用字段包括周期时间 *ptime*、周期标识 *p_id*、数据区 *data*。

(1) 流处理结果流数据包：*stream_pdata*(*sp_id*, *service_id*, *timestamp*, *ptime*, *p_id*, *data*)，其中 *sp_id* 为流处理结果流 ID；

(2) 服务监控流数据包：*monitor_pdata*(*mp_id*, *mon_type*, *service_id*, *timestamp*, *ptime*, *p_id*, *data*)，其中 *mp_id* 表示服务监控数据流 ID、*mon_type* 为监控类型；

(3) 链接资源流数据包：*link_pdata*(*lp_id*, *service_id*, *pic_link*, *pic_info*, *ptime*, *p_id*)，其中 *lp_id* 表示链接资源流 ID、*pic_link* 为图片链接、*pic_info* 为图片描述信息。

2.1.1.2 Redis 缓冲区队列监控表

Redis 缓冲区队列监控表定时记录监控缓冲区队列各指标,其中包括:缓冲区序号 m_id、队列序号 q_id、队列元素个数 $qnum$、队列长度 $qlen$、入队速率 ev、出队速率 dv,如图 4 所示。

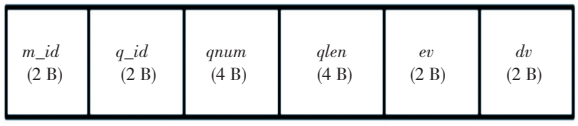

图 4 Redis 缓冲区队列监控表结构

Fig. 4 Redis buffer queue monitoring table structure

2.1.2 Redis 缓冲区队列长度调整

数据处理节点的处理周期为 $T = \{T_1, T_2, \cdots, T_m\}$,Redis 缓冲区队列长度会随若干个周期负载率的不同而发生变化。每个缓冲区队列负载率为 $qload_{ij}$($1 \leqslant i \leqslant m$,$1 \leqslant j \leqslant 3$),计算公式见式(1):

$$qload_{ij} = \frac{qnum_{ij}}{qlen_{ij}} \times 100\% \tag{1}$$

其中,$qnum_{ij}$ 和 $qlen_{ij}$ 分别表示每个缓冲区队列的元素个数和长度,缓冲区队列初始大小 $qlen_{ij}^{\text{init}}$,其计算公式可写为:

$$qlen_{ij}^{\text{init}} = T_i^{\text{init}} * ev_{ij}^{\text{init}} \tag{2}$$

其中,$qlen^{\text{init}} \leqslant qlen_{ij} \leqslant qlen$。每个缓冲区队列的下一周期中,当缓冲区队列负载超过 90%时进行扩容,扩容因子为$\alpha(0 < \alpha < 1)$;当缓冲区队列负载小于 10% 时,缓冲区恢复为初始大小;负载为 10% 到 90% 之间,队列长度不变。此处需用到的公式为:

$$qlen_{ij}^{t+T_i} = \begin{cases} \min(qlen^{\max}, \lfloor (1+\alpha)\, qlen_{ij}^{t} \rfloor), & qload_{ij} > 90\% \\ qlen_{ij}^{t}, & 10\% \leqslant qload_i \leqslant 90\% \\ qlen_{ij}^{\text{init}}, & 0 \leqslant qload_{ij} < 10\% \end{cases} \tag{3}$$

2.2 主动存储中间件实现

2.2.1 缓冲区监控模块

缓冲区监控模块定时收集 Redis 缓冲区队列监控表数据,经计算后更新到 Redis 缓冲区负载表中。Redis 缓冲区负载表为 $RB = \{redis\_buffer_1, redis\_buffer_2, \cdots, redis\_buffer_m\}$,每个子结构见表 1。

表 1 Redis 缓冲区负载表结构

Table 1 Redis buffer load table structure

字段	含义
sort_id	缓冲区综合负载序号(按负载由高到低)
m_id	缓冲区 *id*
ms	综合负载
status	(处理中 - 待处理)

每个 Redis 缓冲区队列入队速率占整体速率的权重为 cw_{ij},权重与入队速率相关,可用式(4)进行计算:

$$cw_{ij} = \frac{ev_{ij}}{\sum_{j=1}^{3} ev_{ij}} \tag{4}$$

其中,ev_{ij} 为每个 Redis 缓冲区队列数据流入队速率。

Redis 缓冲区队列综合负载为 ms_i,由每个队列负载 $qload_{ij}$ 及对应的权重 cw_{ij} 相乘后累加而得,其计算公式见如下:

$$ms_i = \sum_{j=1}^{3} qload_{ij} * cw_{ij} = \sum_{j=1}^{3} \frac{qnum_{ij} * ev_{ij}}{qlen_{ij} * \sum_{j=1}^{3} ev_{ij}} \tag{5}$$

2.2.2 线程读写任务分配模块

当上游的数据速度较快时,Redis 缓冲区队列负载会升高。如果使用单线程处理读写任务,数据存储效率极低。因此,使用线程池的线程读写任务分配方法来解决读写问题。线程池的初始大小设定为 n,n 初始大小为 CPU 核数的 1/2,用来处理 m 个 Redis 缓冲区队列的数据。

线程读写任务分配流程如图 5 所示。由图 5 可知,在主动存储服务启动时,首先初始化线程数量为 n 的线程池,并定时获取缓冲区负载表内容。将综合负载率最高的前 n 个 Redis 缓冲区队列读写任务分配给线程池的 n 个线程,每个线程负责一个 Redis 缓冲区 3 种数据流队列的读写任务。当线程池存在空闲线程时,会优先选择综合负载率最高的 Redis 缓冲区进行处理,将读取到的对象数组批量写入到对应数据库中。

线程读写任务分配算法(Threaded read and write task allocation algorithm,TRWTA)的设计,涉及对线程池的初始化 *ThreadPool*(*n*)、线程回收 *RecycleThread*()、线程唤醒 *WakeupThread*()、线程等待 *WaitThread*() 等函数的调用,具体算法描述如下。

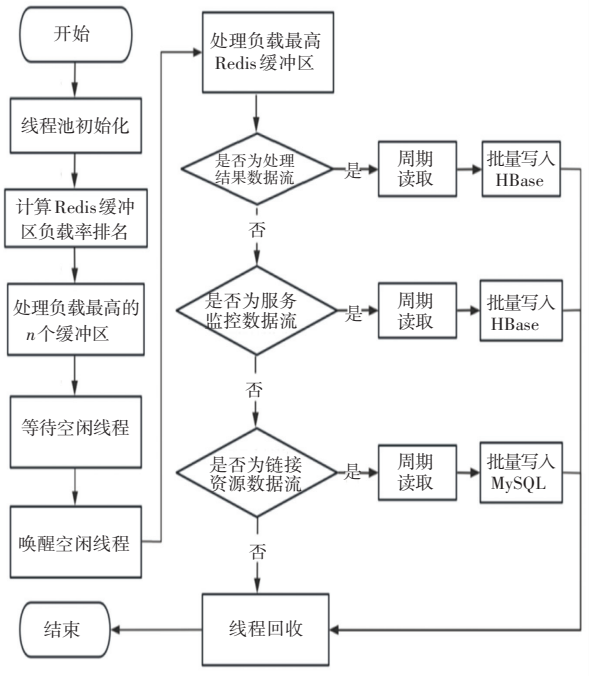

图 5　线程读写任务分配流程

Fig. 5　Flow chart of thread read and write task allocation

算法 1　线程读写任务分配算法 TRWTA

输 入　Redis 缓冲区负载表 $RB = \{redis\_buffer_1, redis\_buffer_2, \cdots, redis\_buffer_m\}$

1.Begin
2.　*ThreadPool*(*n*);
3.　While True:
4.　　计算缓冲区负载;
5.　　处理最高的 *n* 个缓冲区;
6.　　*WaitThread*();
7.　　*WakeupThread*();
8.　　标记该缓冲区为处理中;
9.　　周期读取并批量存储;
10.　　标记该缓冲区为待处理;
11.　　Flush *RB*; //刷新缓冲区负载表
12.　　*RecycleThread*();
13.　End while
14.End

2.2.3　动态线程池模块

固定的线程数不能适应缓冲区负载的变化。线程的数量应考虑整体性能指标，并适时添加或减少线程的数量，以最大程度地利用系统资源和增加存储效率。设定 MS_{avg} 表示 Redis 缓冲区平均负载率，CPU_{avg} 表示平均 CPU 利用率，Mem_{avg} 表示平均内存利用率，其定义分别如下：

$$MS_{avg} = \frac{1}{m}\sum_{i=1}^{m} ms_i \times 100\% \tag{6}$$

$$CPU_{avg} = \frac{1}{m}\sum_{i=1}^{m} cpu_i \times 100\% \tag{7}$$

$$Mem_{avg} = \frac{1}{m}\sum_{i=1}^{m} mem_i \times 100\% \tag{8}$$

其中，cpu_i 和 mem_i 分别表示所在节点的 CPU 和内存利用率。

CPU 核数为 *core_num*，线程阻塞系数为 $k(0 < k < 1)$，线程数 n 约束条件如式(9) 所示：

$$\begin{cases} MS_{avg} \leqslant 0.9 \\ CPU_{avg} \leqslant 0.8 \\ Mem_{avg} \leqslant 0.8 \\ n \leqslant \dfrac{core_{num}}{1-k} \\ 1 \leqslant n \leqslant m \end{cases} \tag{9}$$

定义计算线程池读写压力(Reading and Writing Pressure，*RWP*)，其计算表达式为：

$$RWP = w_1 * MS_{avg} + w_2 * CPU_{avg} + w_3 * Mem_{avg} \tag{10}$$

其中，$w_1 + w_2 + w_3 = 1$ 且 $w_1 > w_3 > w_2$，平均负载率 MS_{avg} 权重最大，平均内存利用率 Mem_{avg} 次之，平均 CPU 利用率 CPU_{avg} 最小。

动态线程池流程如图 6 所示。由图 6 可知，线程池初始化后分配线程执行读写任务，并计算读写压力 *RWP*。当 *RWP* 大于 90% 表明 Redis 缓存区队列数据读写压力过大、存储效率过低，当前线程数量不能满足数据读写任务，应逐步分配线程满足式(9)，分配线程之后进入线程空闲区，等待读写任务的分配；当 *RWP* 小于 10%，线程池读写压力较低，表明有较多处理线程空闲得不到有效利用，此时回收空闲线程数量的一半，以减少线程资源浪费。

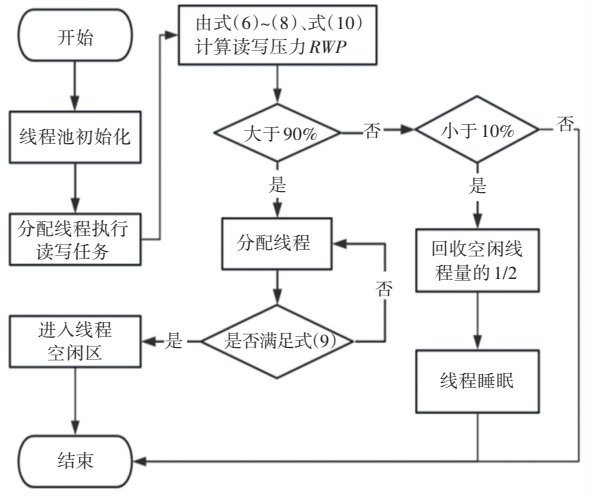

图 6　动态线程池流程图

Fig. 6　Dynamic thread pool flowchart

为避免系统频繁分配线程带来的不稳定性,采用平均周期时间 T_{avg} 作为动态线程池调整间隔,其计算公式如下:

$$T_{avg} = \frac{1}{m}\sum_{i=1}^{m} T_i \tag{11}$$

其中,T_i 为每个上游数据节点的周期。动态线程池算法的伪代码描述如下。

算法 2 动态线程池算法

输入 线程数量 n

1.Begin

2. While True:

3. 计算读写压力 *RWP*;// 利用式(6) ~(8)、式(10) 计算读写压力 *RWP*

4. If *RWP* 大于 90%

5. 分配线程;

6. 进入线程空闲区;

7. Else

8. 回收空闲线程数的一半;

9. sleep (T_{avg}) //睡眠时间为平均周期;

10. End while

11.End

2.3 存储层

2.3.1 Rowkey 设计与预分区

在 HBase 设计 RowKey 时需满足唯一性、散列性、简短性等。本文采用 MD5 (Message Digest Algorithm 5)算法确保良好散列和唯一性,以此分散数据到不同分区避免写入热点。Rowkey 按 ASCII 排序,通过 *Long.MAX_VALUE - Timestamp* 把时间戳处理为递减时间戳,来确保最新数据优先查询,提高实时查询效率。研究设计的 RowKey 分述如下:

(1)流处理结果数据 Rowkey 设计

MD5(*service_id* + 小时)%16 + *equip_id* + (*Long.MAX_VALUE - Timestamp*) + *sd_id*

(2)服务监控数据 Rowkey 设计

MD5(*service_id* + *md_type* + 小时)%24 + *service_id* + (*Long.MAX_VALUE - Timestamp*) + *md_id*

HBase 表初始分区由一个默认分区组成。为避免数据量不断增大导致频繁的分区拆分,影响写入性能,所以进行预分区。由于 RowKey 与时间相关,高峰期在每天 6:30 到 8:30、11:30 到 13:30、15:00 到 19:30,其他时间数据平稳。为分散小时数据,设置分区号为 *MD5*(*service_id* + 小时)%16,分散到 16 个分区;服务监控数据较稳定,分区号为 *MD5*(*service_id* + *md_type* + 小时)%24,分散到 24 个分区。

2.3.2 定期删除

持续的数据存储很可能会造成存储资源浪费,可定期删除时间超过某阈值的数据。HBase 对每行数据添加列属性 Timestamp 后,设置单表的存活时间(Time To Live, *TTL*)。

在 HBase 合并时,系统会判断当前时间是否大于等于数据生成时间加上存活时间 *TTL*, 由此保证集群存储的正常运行。

2.4 数据查询

查询时序如图 7 所示。由图 7 可知,根据查询接口构造查询条件,若数据为热点则直接返回,否则查询数据库;对于热点数据,查询后回填缓存,最后把封装查询结果返回客户端。

处理结果数据、服务监控数据和资源链接数据的查询接口设置见表 2。

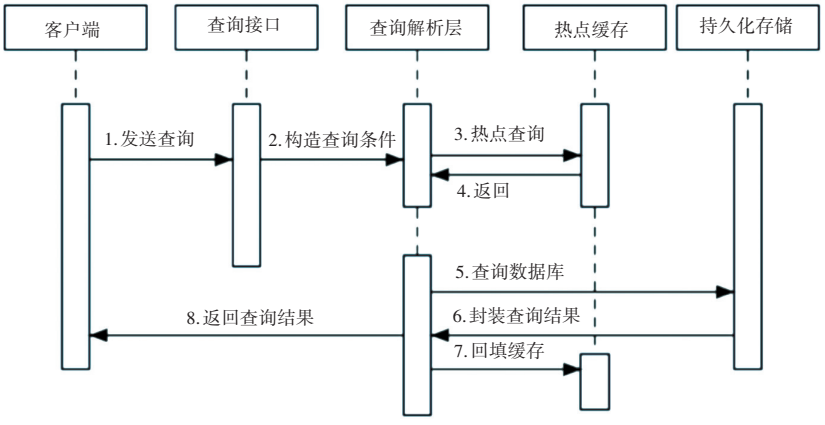

图 7 数据读取时序图

Fig. 7 Data reading sequence diagram

表 2　查询接口示意表

Table 2　Query interface diagram

软　件	接　口	类　型	说　明
流处理结果数据	/stream/query/{*service_id*}/[{*sd_id* =} | {*start_time* =} | {*end_time*}]	Get	服务节点 ID、流处理结果流 ID 开始/结束时间
服务监控数据	/monitor/query/{ service_id}/[{*md_type* =} | {*mp_id* =} | {*start_time* =} | {*end_time*}]	Get	服务节点 ID、监控类型、服务监控 ID、开始/结束时间
资源链接数据	/meta/query/{*service_id*}/[{*lp_id* =}…]	Get	服务节点 ID、链接资源流 ID

3　实验与分析

3.1　实验环境

实验数据采用某城市公交车内的实时监控图片流，通过分流交给多个数据处理节点，形成并发数据流作为实验对象。操作系统采用 8 台 CentOS Linux release 7.9.2009 (Core)服务器，核数为 8，硬件配置见表 3，以此部署集群环境。

表 3　硬件配置表

Table 3　Hardware configuration table

硬件	配置
CPU	AMD Opteron(tm) Processor 6320
内存	16 G
硬盘	40 G

软件配置见表 4。表 4 中，Hadoop 副本数设置为 3。

表 4　软件配置表

Table 4　Software configuration table

软件	版本号
Hadoop	3.1.3
Zookeeper	3.5.7
HBase	2.0.5
JDK	Java HotSpot(TM) 64-Bit Server VM (build 25.212-b10, mixed mode)
Redis	6.0.8

3.2　分析指标

为了评估多个 Redis 缓冲区的变化，引入标准差以衡量其稳定性，将负载区的标准差定义为 *MD*，数学定义公式为：

$$MD = \frac{1}{m}\sqrt{\sum_{i=1}^{m}(ms_i - MS_{\text{avg}})^2} \tag{12}$$

其中，ms_i 为每个 Redis 缓冲区的负载率，MS_{avg} 为 Redis 缓冲区负载的平均值。

数据库的存储性能，使用存储吞吐量 F 来衡量，存储吞吐量越大表示存储性能越高，F 计算公式如下：

$$F = \sum_{i=1}^{m}\frac{thread_{\text{num}} * req_{\text{num}}}{total_{\text{time}}} \tag{13}$$

其中，$thread_{\text{num}}$、req_{num} 分别表示线程数量和每个线程周期批量存储的数据量，$total_{\text{time}}$ 为每个缓冲区一个周期所消耗的时间。

数据流从处理后到存储经过一个周期的平均存储时延为 *TD*，其计算公式见式(14)：

$$TD = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{3}(t_{ij}^{(1)} + t_{ij}^{(2)} + t_{ij}^{(3)}) \tag{14}$$

其中，$t_{ij}^{(1)}$、$t_{ij}^{(2)}$、$t_{ij}^{(3)}$ 分别表示每个缓冲区队列的入队延时、等待延时和处理时延。研究可知，$t_{ij}^{(2)}$ 为等待延时，可由式(15)运算求出：

$$t_{ij}^{(2)} = \frac{qnum_{ij}}{ev_{ij} - dv_{ij}} \tag{15}$$

其中，$qnum_{ij}$、ev_{ij}、dv_{ij} 分别表示进入 Redis 缓冲区队列的元素个数、入队速率与出队速率。

3.3　实验过程和分析

实验缓冲区队列的扩容因子 α 设置 0.2，周期时间设置为 5 s，缓冲区队列的初始长度 $qlen^{\text{init}}$ 为 250，$qlen^{\max}$ 为 1 000。

3.3.1　Redis 缓冲区分析

为验证本文所提面向并发数据流的性能，在固定速率图片数据流(每秒 50 条数据)发送给数据处理节点加以处理。随着 Redis 缓冲区的增多，存储吞吐量、平均 CPU 利用率和 Redis 缓冲区平均负载的情况，如图 8 所示。

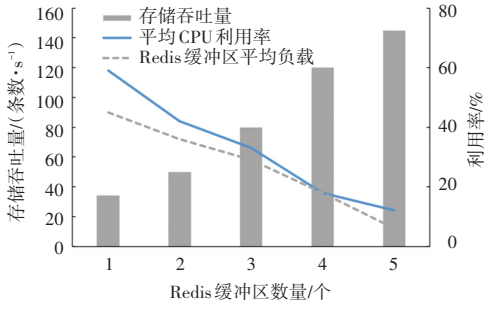

图 8 Redis 缓冲区分析

Fig. 8 Redis buffer analysis

实验结果表明,Redis 缓冲区数量与存储吞吐量成正比,与平均 CPU 利用率和 Redis 缓冲区平均负载成反比。当缓冲区负载不断升高时,需要占用更多线程处理读写任务,线程在运行与调度中所消耗的资源与 CPU 资源成正相关;当缓冲区负载降低时,线程的读写任务得到缓解,CPU 利用率呈下降趋势。实验验证了本文所提 Redis 缓冲区的有效性,在降低缓冲负载的同时,提高了存储吞吐量。

3.3.2 主动存储中间件分析

(1)线程读写任务分配算法(TRWTA)分析。对比使用与未使用 TRWTA 算法在连续 10 个周期处理下,Redis 缓冲区平均负载率的变化,实验结果如图 9 所示。

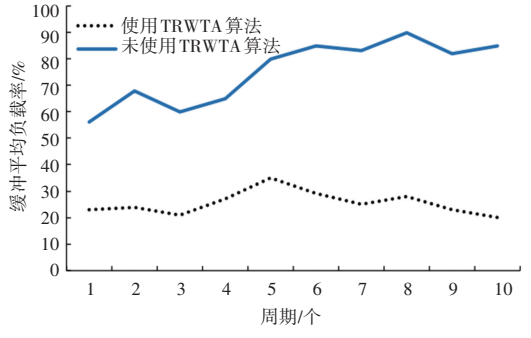

图 9 使用和未使用 TRWTA 算法的对比

Fig. 9 Comparison of the results with and without using TRWTA algorithm

由图 9 可见,使用 TRWTA 算法能够优先处理负载量高的 Redis 缓冲区,有效缓解了负载,标准差稳定在 0.75~1.30 之间。未使用 TRWTA 时,缓冲负载率较高,标准差波动不稳定,范围在 1.5~3.8 之间。实验表明,TRWTA 算法可提高存储接入的效率,及时处理 Redis 缓冲区负载率高的数据。

(2)固定线程与动态线程池对比。通过实验对比使用固定线程和动态线程池在处理读写任务时 *RWP* 变化,其中 w_1、w_2、w_3 分别为0.45、0.20 和 0.35,阻塞系数 k 为 0.2,实验结果如图 10 所示。

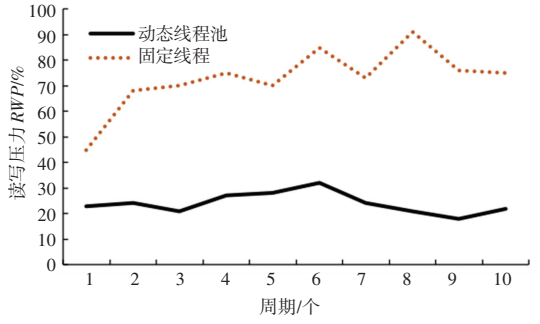

图 10 固定线程与动态线程池的读写压力对比

Fig. 10 Comparison of *RWP* between fixed thread and dynamic thread pool

由此可见,动态线程池较固定线程在读写压力 *RWP* 上明显降低。当 *RWP* 大于 90% 时,通过分配线程缓解线程数量不足;当 *RWP* 小于 10% 时,通过回收空闲线程数量的一半减少线程资源浪费,降低了读写压力。

3.3.3 方案对比

为验证所提并发数据流处理的存储接入方法性能,将其与传统存储方法进行对比。传统存储方法是在数据处理后直接存储相应的数据库,并未考虑周期存储、线程池分配等情况。部署 5 个 Redis 缓冲区,图片流速率为周期 5 s,图片速率分别为每秒 20、50、100、150、200,实验指标选取 10 个周期下的平均存储时延 *TD*,通过每组速率下 10 个周期的平均值作为存储时延,如图 11 所示。

分析图 11 可知,本文所提面向并发数据流的存储接入方法,在平均存储时延上较传统存储方法得到明显改善。随着数据流速率的不断提高,本文方法充分利用 Redis 缓冲区与主动存储中间件结合,改善了内存利用率,减少了内存资源的消耗,提高了存储的效率。

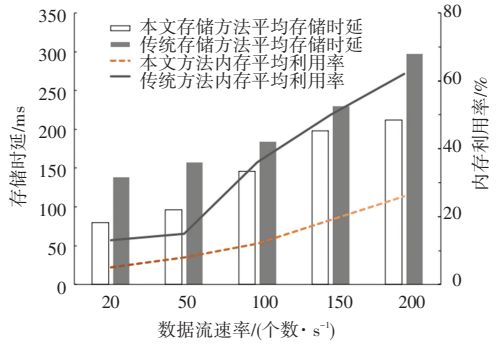

图 11 存储方法结果对比图

Fig. 11 Comparison of storage methods

4　结束语

本文提出的面向并发数据流处理的存储接入方法，从多个数据处理节点接收周期数据流，并为每个数据处理节点构建缓冲区，缓冲区设置多个队列缓冲上游节点产生的不同数据流，使用数据流协议规范传输各种数据流。构建主动存储中间件，利用缓冲负载表定时更新缓冲区多个队列的综合负载，有利于根据综合负载进行读写任务的调整和线程动态分配。使用线程读写任务分配算法 TRWTA 优先处理综合负载高的缓冲区以加快数据传输，通过周期轮转读取与批量写入提高存储效率。使用动态线程池计算读写压力 *RWP* 动态增加或减少线程，提高存储的稳定性。

经实验表明，通过 Redis 缓冲区缓解系统负载的同时提高了存储吞吐量；使用 TRWTA 算法减轻了缓冲高负载区数据阻塞，提高了读写效率；动态线程池较固定线程明显降低读写压力，本文所提存储方法较传统存储方法有效降低存储时延和平均内存利用率，保证了数据库的稳定性。

参考文献

[1] CAO Keyan, LIU Yefan, MENG Gongjie, et al. An overview on edge computing research[J]. IEEE Access, 2020, 8: 85714-85728.

[2] QIU Tie , CHI Jiancheng, ZHOU Xiaobo, et al. Edge computing in industrial Internet of Things: Architecture, advances and challenges[J]. IEEE Communications Surveys & Tutorials, 2020, 22: 2462-2488.

[3] 涂聪，陈庆奎. 面向 AI 数据流处理的边缘 GPU 集群通信系统[J]. 小型微型计算机系统，2022，43(6)：1147-1153.

[4] 黄东生，陈庆奎. 一个并发 AI 数据流处理节点内的通信模型[J]. 智能计算机与应用，2022，12(11)：26-33，40.

[5] KOLAJO T, DARAMOLA O, ADEBIYI A. Big data stream analysis: A systematic literature review[J]. Journal of Big Data, 2019, 6(1): 1-30.

[6] 陆铭琛，吕晏齐，刘睿诚，等. 基于水车模型的时序大数据快速存储[J]. 计算机科学，2023，50(1)：25-33.

[7] 陆婷，房俊，乔彦克. 基于 HBase 的交通流数据实时存储系统[J]. 计算机应用，2015，35(1)：103-107，135.

[8] FREIRE D L, FRANTZ R Z, ROOS-FRANTZ F. Towards optimal thread pool configuration for run-time systems of integration platforms[J]. International Journal of Computer Applications in Technology, 2020, 62(2): 129-147.

[9] SCHMID M, MOTTOK J. Response time analysis of parallel real-time dag tasks scheduled by thread pools[C]//29th International Conference on Real-Time Networks and Systems. New York, USA: ACM, 2021: 173-183.

[10] NGUYEN H T, DO T V, ROTTER C. Optimizing the resource usage of actor-based systems[J]. Journal of Network and Computer Applications, 2021, 190: 103143.

[11] 赖德军，陈建华，罗娟. 一种通用启发式线程池的设计与实现[J]. 网络安全技术与应用，2010 (12)：52-54.

[12] 梁少林. 基于 LSM 树的云存储数据差异性存储节能优化算法[J]. 吉林大学学报(信息科学版)，2022，40(2)：282-287.

[13] HASSAN M U, YAQOOB I, ZULFIQAR S, et al. A comprehensive study of HBase storage architecture-A systematic literature review[J]. Symmetry, 2021, 13(1): 109.

[14] WANG Keliu, ZHAI Guolin, WANG Min, et al. Building an efficient storage model of spatial-temporal information based on HBase[J]. Journal of Spatial Science, 2019, 64(2): 301-317.

[15] 杨力，陈建廷，向阳. 基于 HBase 的工业时序大数据分布式存储性能优化策略[J]. 计算机应用，2023，43(3)：759-766.

[16] ZHANG Jingbin, MA Meng, WANG Ping, et al. Middleware for the Internet of Things: A survey on requirements, enabling technologies, and solutions[J]. Journal of Systems Architecture, 2021, 117 : 102098.

[17] PRADEEP P, KRISHNAMOORTHY S, VASILAKOS A V, et al. A holistic approach to a context-aware IoT ecosystem with adaptive ubiquitous middleware[J]. Pervasive & Mobile Computing, 2021, 72: 101342.

[18] 类兴邦，房俊. 基于融合数据库的海量传感器信息存储架构[J]. 计算机科学，2016，43(6)：68-71，111.

[19] 马书群. 船舶大数据信息中心的高性能存储架构设计[J]. 舰船科学技术，2020，42(2)：148-150.

[20] 林琳. 开发边缘计算存储需要考虑的关键因素[J]. 计算机与网络，2021，47(10)：43.