文章编号:1001 - 9081(2008)04 - 0877 - 04

基于动态反馈多线程的大数据驱动缓存的设计与实现

林建素,钟 勇,丁 洁
(中国科学院 成都计算机应用研究所,成都 610041)
(yuehuancheng@126.com)

摘 要:对大数据驱动缓存设计原理进行了研究,针对常规数据传递策略时间性能和在性能稳定性上的不足,在大数据的驱动缓存设计上,提出了基于动态反馈的多线程主动等待策略。该策略首先采用主线程主动等待的方式实现了多线程数据处理方案,在此基础上引入动态反馈,通过平衡数据驱动和服务器、应用系统之间数据流与指令流的传递,以提供常规多线程传递策略无法实现的负载不确定性下的性能保证。

关键词:传递策略;动态反馈;多线程;负载不确定性
中图分类号: TP311.13;TP392 **文献标志码**:A

Design and implementation of large data buffer based on multithreading using dynamic feedback

LIN Jian-su, ZHONG Yong, DING Jie
(*Chengdu Institute of Computer Application, Chinese Academy of Sciences, Chengdu Sichuan* 610041, *China*)

Abstract: The designing principle of large data driver buffer was studied. Ordinary data transfer strategies have their limitations either in time efficiency or performance stability. A multithreading positive waiting strategy based on dynamic feedback was proposed. Firstly, multithreading data transfer scheme was implemented in the form of main thread's positive wait. For high performance guarantees under unpredictable workloads' condition, dynamic feedback was imported to balance the data and instruction transfer between server-to-driver and driver-to-client.

Key words: transfer strategy; dynamic feedback; multithreading; unpredictable workloads

0 引言

大数据的传递性能显得日益重要,常成为数据库应用系统效率的瓶颈所在。设计大数据缓存的目的是在大型应用系统中平衡并加速数据驱动与服务器、应用系统之间的数据流和指令流的执行,其中数据流的传递策略是决定驱动性能的关键。

本文从数据驱动和缓存的原理出发,分析大数据传递过程中的关键性能决定要素。在建立典型的大数据传递假设模型的基础上,通过分析现存多种传递策略的不足,提出了以多线程请求方等待策略,并计算其理想时间效率。在此基础上,分析该策略过程中网络、环境等导致的负载不确定性,提出了通过动态反馈调整多线程执行时间片的方式,以尽可能向估值中的时间性能靠近。最后通过实验数据验证负载不确定性下该策略的性能稳定性和实际效果。

1 数据驱动的缓存原理

1.1 数据驱动

在数据库应用系统中,与数据库服务器之间至少存在如下交互:

1)数据的检索,即应用系统检索数据库服务器获取所需数据的过程;

2)数据的提交,即应用系统提交数据到数据库服务器的过程;

3)数据库指令的执行。

由于操作系统平台与应用系统平台多样性的存在,数据库服务器上的数据组织结构和指令语法无法直接适应各类应用系统的需求,在如上的交互过程中,将不可避免地需要进行相应的转化。基于实现通用性的考虑,应建立一个能够处理常用操作系统和应用系统的数据驱动层。

数据驱动这个中转站出现后,数据、指令的传递都需经历如下三个阶段:

1)服务器与数据驱动之间的数据传递;

2)数据驱动进行数据组织结构和指令格式的转化;

3)数据驱动与应用系统之间数据的传递。

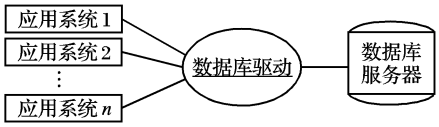

图1 数据驱动层

1.2 驱动的缓存原理

信息日新月异,需要处理的数据量也变得日益庞大,能否快速处理大数据已经成为评判数据库系统优劣的关键因素,也是数据库系统的瓶颈所在。

加入数据驱动层后,数据的处理同样需要经过图1所示的三个步骤,由于数据组织结构的不一致,需要建立适当的缓存,以向请求方快速传递所需的数据。另外,由于请求一方存在多次相关请求,以缓存的方式保存被请求方一次性传送的所有相关信息,将极大地减少通信次数和数据库服务器负载

收稿日期:2007 - 10 - 30;**修回日期**:2007 - 12 - 07。 **基金项目**:四川省应用技术研究与开发项目(2007Z01-019)。

作者简介:林建素(1982 -),男,浙江温州人,系统分析师,硕士研究生,主要研究方向:数据库驱动设计、软件过程与项目管理; 钟勇(1966 -),男,四川岳池人,研究员,博士生导师,主要研究方向:数据仓库技术、软件工程与过程方法技术; 丁洁(1982 -),女,重庆江津人,硕士研究生,主要研究方向:智能软件。

量;并且,应用系统的再次请求只需要与数据组织结构一致的数据驱动进行交互。

2 大数据传递策略分析

在请求方发出数据请求后,返回的数据需要经过两个阶段的传递和数据库驱动中数据组织结构的转化。在此过程中,数据集是最常用的数据类型,大数据集的传递性能将决定整个系统的运行效率,其基本结构可分为:数据集描述部分、以行和单元格为基本单位的实际数据。

2.1 大数据模型假设

大数据的传递,其驱动设计上的时间性能,可以不用数据集描述部分和命令分析部分的执行时间,现以"行"为单位建立如下假设:

1)数据从数据库服务器到数据驱动的平均时间为 $T1$,数据在驱动中的平均转化时间为 $T2$,数据从驱动传递到应用系统并使用的平均时间为 $T3$。另外,$T2 \ll T1$,$T2 \ll T3$,但不可忽略,$T1$ 和 $T3$ 的大小无法比较,可能 $T1$ 大,亦可能 $T3$ 大。

2)忽略数据描述部分在各个过程中的传递时间;忽略请求指令的执行过程。

3)大数据的行数为 N,发送请求的开始时刻为 0,请求方处理完数据的时刻为 Te。

2.2 数据的传递策略

目前存在几种比较常见的数据集传递策略:

1)完全黑盒模块化策略

这种策略下,请求方发送一个数据请求命令,该请求传递到数据驱动后,数据驱动重新封装该命令并传递到数据库服务器,数据库服务器获取请求后返回指定的数据给数据驱动,而数据驱动得到服务器返回的数据后,封装该数据的所有内容为请求方可直接使用的数据缓存,并填入所有数据,最后把该数据缓存对象返回给请求方。

这种策略实现非常简单,能够非常清晰划分各模块职责,然而其对大数据的传送处理效果并不理想,其处理完数据的时刻为:

$$Te = (T1 + T2 + T3) \times N \tag{1}$$

另外,由于在数据驱动返回数据缓存对象给请求方之前,请求方处于停滞等待状态。

2)逐行处理策略

这种策略下,驱动将在每获取一行数据后,即向请求方返回,并等待请求方对获取的数据进行处理后,才继续接收后面的数据。这种策略是对第一种策略的修正,其主要目的是分散三部分占用的 CPU 执行时间,其处理完数据的时刻仍然为:

$$Te = (T1 + T2 + T3) \times N \tag{2}$$

2.3 数据传递的多线程策略

在数据传递过程中,不难发现,$T1$、$T2$、$T3$ 所对应的三个过程是可以同步进行的,所以考虑采用多线程的策略。

下面通过分别为三个过程建立各自线程的方式实现,设建立的线程分别是 Thread1,Thread2,Thread3。由于这里的 Thread1、Thread2 为驱动中的线程,而 Thread3 为应用请求方的线程,必须以请求方主动等待的方式运行,所以这种策略又叫多线程请求方等待策略。

其运行将按照如下方式进行:

1)由于 Thread3 线程是数据请求主线程,所以从 0 时刻开始就已经开启并运行,并且 Thread3 线程接收完数据的时刻即为 Te,我们的目标是尽量使 Te 变小;

2)当驱动开始接收数据库服务器发送的数据时,Thread1 开始运行,直至数据接收完毕,其中在每次接收完一行数据时开启 Thread2 线程;

3)Thread2 线程负责对当前行数据组织结构进行转化,转化结束后立即暂停自己的运行;

4)Thread3 线程将循环等待驱动缓存中的数据,只要有新数据到达(即 $T2$ 执行完毕),马上进行 $T3$ 的处理。

图 2 演示了线程的执行过程。

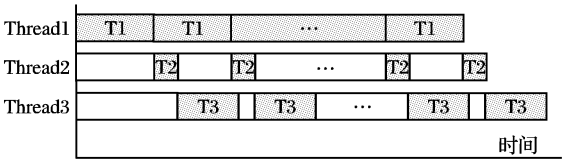

图 2 多线程的执行序列

图 2 中有 N 行数据,并假设处理器分配了足够的时间片,则 Thread1 和 Thread2 的执行总时间为 $T1 \times N + T2$,由于 $T1$ 和 $T3$ 的大小无法比较,Thread3 的执行总时间存在如下情况:

如果 $T1 \geqslant T3$,则 Thread3 的执行总时间为 $T1 \times N + T2 + T3$,这时,处理完数据的时刻为:

$$Te = T1 \times N + T2 + T3 \tag{3}$$

如果 $T1 < T3$,则 Thread3 的执行总时间为 $T1 + T2 + T3 \times N$,这时,处理完数据的时刻为:

$$Te = T1 + T2 + T3 \times N \tag{4}$$

这两种可能性相同,所以采用多线程请求方等待策略后,处理完数据的平均时刻为:

$$Te = (T1 + T3) \times (N + 1)/2 + T2 \tag{5}$$

如果把 $T1$ 和 $T3$ 看成同数量级的,则相比前面两种策略,所用时间比为:

$$\frac{Te(5)}{Te(1)} = \frac{(T1 + T3) \times (N + 1)/2 + T2}{(T1 + T2 + T3) \times N} \approx 1/2 \tag{6}$$

采用多线程请求方等待策略相当于效率提升了约 50%,但这是理想的状态,即忽略了处理器分配给 Thread1 和 Thread3 时间片多少的不确定性,并且 $T1$ 和 $T3$ 可能存在较大的差异。为了保证这种效率,下面将引入控制线程的时间片分配调度策略。

3 动态反馈多线程调度

由于大数据传递过程中的 T1 和 T3 属于未预料的时间负载,无法预先获知,并且在传递过程中由于环境、网络等因素而发生动态变化,所以,在默认情况下,处理器对这两个线程进行时间片分配方案无法实现其最优的时间性能。

3.1 动态反馈调度原理

在通常情况下,对 Thread1 和 Thread3 的分配与调度上采用的是主线程等待策略,即让 Thread3 等待 Thread1 的执行,在此过程中,处理器可能在做无用功;另外,如果出现 T1 < T3 的情况,也并不会让 Thread1 先停下,而把处理器的时间分配给当前更需要时间处理的 Thread3。分析其实质,实际上是处理器在调度 Thread1 和 Thread3 时,为其假想了优先级和负载量,然后分配相应的时间片,这是一种开环且静态的调度策略。

静态的调度策略有其明显的缺点,因为优先级已经默认确定后,运行时刻无法修改,即使出现线程等待执行的情况,也不会灵活地进行调度。

相反地,动态的调度策略可以不用事先设定线程优先级,也不用设定线程的负载量,它只能获知当前的负载量。但它可以在动态运行的过程中不断调整处理器的当前优先级进行灵活的时间片分配。

3.2　动态反馈多线程调度实现

这里将引入动态的反馈调度策略,通过动态控制 Thread1 和 Thread3 分配的时间片数目来减少 Thread3 等待的时间,平衡 Thread1 和 Thread3 实际占用处理器的时间,最终达到式(6)的性能。

为讲述方便,下面将引入一些新的假设:

1)在单位时间里面,处理器能够分配给 Thread1 和 Thread3 的时间片总和为 M,即 M 的时间片完全由 Thread1 和 Thread3 使用;

2)能够动态获取最新 Thread1 和 Thread3 在行数据传递所需的时间,即 T1 和 T3 是动态可变的,每次接收前记录最后一次的 T1 和 T3 时间,设分别需要 M1 和 M3 的时间片处理;

3)Thread3 作为主线程能够控制单位时间分配给子线程 Thread1 的时间片数目,并控制子线程之间的优先级。

有了如上假设后,为了实现动态的反馈调度策略,需要在前面的多线程请求方等待策略中加入以下新的策略:

在每次 Thread1 开始接收新的一行数据时,首先根据最新一次 M1 和 M3 值,按照相应比例分配时间片,并使得 Thread1 的优先级高于 Thread3,并记录当前行数据传递过程中的实际 M1 和 M3 值。

动态反馈多线程的调度算法伪代码如下所示:

```
Thread1{                              //Thread1 线程,由 Thread3 触发
  void run(){
    getData();
  }
  //服务器到驱动的数据传递过程
  void getData(){
    while(new row exists){
      getRowData();
      //每接收完一行数据后,启动 Thread2 线程
      Thread2.run();
    }
  }
}
Thread2{                              //Thread2 线程,由 Thread1 多次触发
  void run(){
    //为每行数据进行数据组织结构转化
    translateRowData();
    this.stop();
  }
}
Thread3{                                    //Thread3 线程,为主线程
  void run(){
    Thread1.run();                          //启动 Thread1 线程
    while(data Transferring){
      getTime(M1,M3);                       //获取最新 M1,M3 数据
      //分别为 Thread1 和 Thread3 线程分配时间片
      allocateTime(Thread1,M1);
      allocateTime(this,M3);
      //设置优先级;
      setPriority(Thread1 bigger than Thread3);
      //处理数据 包括接收驱动数据和使用
      handleRowData();
      calculate(M1,M3);                     //计算新的 M1,M3 值
    }
  }
  void handleRowData(){
    while(no data from Thread2){
      //无数据时需要循环等待
      continue();
    }
    getDataFromDriver();
    applicateData();
  }
  void allocateTime(Thread,time);
}
```

这样修改,实际上并没有修改原先的多线程执行策略,但是由于单位时间内能够分配适当的时间片给 Thread1 和 Thread3,因而出现如下情况后,不会浪费处理器时间:

1)当 M1 > M3 时,因为单位时间分配给 Thread1 的时间片变长,从而 Thread3 在被分配给时间片时不至于等待;

2)当 M1 < M3 时,也可以保证在 Thread3 在处理完行数之前,不会出现 Thread3 来不及处理 Thread1 传递的每行数据。

3.3　实验测试与分析

下面将通过对一个实例模型的测试来分析算法实际的运行时的性能。创建测试实例的过程如下:

首先,创建供测试的服务器数据,为了保证大数据,选用 Blob 类型作为主要测试数据,创建的数据表如表 1。

表 1　创建的数据表

列名	类型	其他
Id	int	主键
picture	Blob(image)	测试大小:1 MB ~ 10 MB

在该表中插入 20 行实例数据,保证 picture 列能够随机满足其大小的要求。

其次,令创建 Thread1 和 Thread3 总共分配的时间片基数为 100,并在最开始测试时设定 $M1$ 和 $M3$ 分别为 50,在对行数据接收的过程中,$M1$ 和 $M3$ 将不断发生变化,需要记录每行数据传输过程的实际 $T1$ 和 $T3$。

最后,为了提高 Thread3 对接收到数据的处理复杂度的随机化,将对 picture 中的图片文件进行压缩处理,并且由于不同的图片及其大小能够保证压缩所需时间的不一致。

表 2　实例测试结果

数据行	$M1$	$M3$	$T1$ /ms	$T3$ /ms	测试操作
1	50	50	120	70	
2	63	37	80	78	
3	64	36	56	50	
4	68	32	200	220	
5	64	36	32	35	
6	61	39	43	50	
7	57	43	60	53	
8	60	40	134	126	
9	62	38	140	128	
10	65	35	50	80	提高数据处理复杂度
11	50	50	43	54	
12	44	56	78	86	
13	42	58	139	142	
14	42	58	80	79	
15	42	58	90	87	
16	43	57	278	260	
17	44	56	34	29	
18	48	52	78	56	降低数据处理复杂度
19	55	45	102	87	
20	60	40	120	80	

根据测试数据,可计算 $Te(1)$ 和 $Te(5)$ 的值分别如下:

$$Te(1) = \sum_{i=0}^{20} (T1(i) + T2(i)) = 3807\ ms$$

$$Te(5) = \sum_{i=0}^{20} max(T1(i), T2(i)) = 2039\ ms$$

所以，$\frac{Te(5)}{Te(1)} = \frac{2039}{3807} \approx 53.56\%$，接近于式(6)中50%的最理想状态。

测试分析：由表1中的实验测试结果可知，接收第1～9行时，Thread1比Thread3所需时间多，于是根据调度算法分配了更多的时间片，并且在Thread1和Thread3的接收和应用模式不发生大变化的情况下，*M*1和*M*3能够保持一个动态的相对稳定性；而在接收第10行开始，为了模拟负载的不确定性变化，人为地提高了数据处理的复杂度，于是Thread3需要获得更多的时间片，根据调度算法，*M*1和*M*3也动态发生变化，并逐渐稳定；同样地，在接收第18行数据开始，由于人为降低了数据处理的复杂度，于是*M*1和*M*3的分配再次恢复到给Thread1多分配时间片的状态。

由实验数据及其分析可以看出，在采用了主动等待多线程大数据传递策略后，能够通过调度保证负载不确定性下的性能。

4 结语

由于数据驱动中间层的出现，缓存中大数据传递的时间性能显得日益重要，但现存的策略无法解决负载不确定性上性能的保证。在此前提下，本文提出了主动等待多线程的大数据传递策略，基于负载不确定性的考虑，在此基础上，提出了通过动态反馈控制多线程时间片分配的方案。实验数据标明：该方案能够实现在负载不确定性下的性能稳定性，实验结果数据接近于算法分析的理想状态。

参考文献：

[1] O'NEIL P, O'NEIL E. Database-Principles, Programming, and Performance[M]. USA: Morgan Kaufmann, 2002.

[2] RIORDAN R M. Designing effective database system[M]. USA: Prentice Hall, 2006.

[3] SHASHA D, BONNET P. Database Tuning: Principles, Experiments, and Troubleshooting Techniques[M]. USA: Morgan Kaufmann, 2004.

[4] AKHTER S, ROBERTS J. Multi-Core Programming: Increasing performance through software multi-threading[M]. USA: Intel Press, 2007.

[5] CULLER D E. Parallel Computer Architecture[M]. USA: Morgan Kaufmann, 2005.

[6] BIC L, GAO G R, GAUDIOT J-L. Advanced Topics in Dataflow Computing and Multithreading[M]. Los Alamitos, CA, USA: IEEE Computer Society Press, 1995.

[7] LU CHENYANG. Feedback control real-time scheduling: Framework, Modeling, and algorithms[M]. USA: University of Virginia, 2002.

[8] SKOGESTAD S, POSTLETHWAITE I. Multivariable Feedback Control: Analysis and Design[M]. New York, USA: John Wiley & Sons, 2001.

(上接第871页)

3.2 内存限制影响的比较

在本实验中，|*AList*|和|*DList*|的大小分别为21750和69969。图7描述了内存大小/数据集大小的值与结构连接算法的运行时间之间的关系，从而体现出DRIAM算法和XR-Tree算法在内存受限情况下的性能。

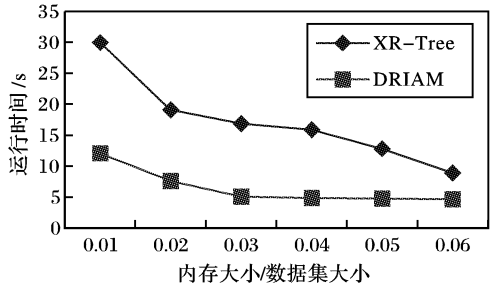

图7 内存限制对DRIAM算法和XR-Tree算法的影响比较

从图7可以看出，当内存大小受到限制时，XR-Tree算法的性能急剧下降；而DRIAM算法相对而言保持良好的性能。因此，DRIAM算法可以高效地应用在不同内存限制的环境中。

4 结语

本文集中讨论了Native XML数据库的结构连接算法，提出了一种新的结构连接算法：基于深度均匀划分的结构连接算法DRIAM。理论分析和实验结果表明，该算法不要求输入数据*AList*和*DList*有序或在其节点编码上建有索引，避免了排序和索引所增加的额外开销；该算法不需要输入数据*AList*和*DList*全部加载到内存中，可以适应不同内存大小限制的情况，并且该算法时间复杂度非常低。这是在Native XML数据库结构连接算法的研究上进行的有效尝试。

本文提出的结构连接算法I/O访问次数较多，如何降低由此带来的代价，如何降低偏斜数据对该算法的影响，等等，都是值得进一步研究的问题。

参考文献：

[1] 冯建华，钱乾，廖雨果，等. 纯XML数据库研究综述[J]. 计算机应用研究，2006，23(6)：1－7.

[2] ZHANG CHUN, NAAGHTON J, DeWITT D, *et al*. On supporting containment queries in relational database management systems [C]// ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2001: 426－437.

[3] AL-KHALIFA S, JAGADISH H V, KOUDAS N, *et al*. Structural joins: a primitive for efficient XML query pattern matching[C]// 18th International Conference on Data Engineering. San Jose: IEEE Computer Society, 2002: 141－152.

[4] CHIEN S Y, VAGENA Z, ZHANG DONG-HUI, *et al*. Efficient structural joins on indexed XML documents[C]// 28th International Conference on Very Large Data Bases. Hong Kong: Morgan Kaufmann Publishers, 2002: 263－274.

[5] JIANG HAI-FENG, LU HONG-JUN, WANG WEI, *et al*. XR-Tree: indexing XML data for efficient structural joins[C]// 19th International Conference on Data Engineering. Los Alamitos: IEEE Press, 2003: 253－264.

[6] 王静，孟小峰，王珊. 基于区域划分的XML结构连接[J]. 软件学报，2004，15(5)：720－729.

[7] WIRTH N. Type extentions[J]. ACM Transaction on Programming Languages and systems, 1988, 10(2): 204－214.

[8] 冯建华. 纯XML数据库的查询求解关键问题研究[D]. 北京：清华大学，2006.

[9] Xmark: An XML Benchmark Project[EB/OL]. [2007－08－01]. http://monetdb.cwi.nl/xml.