内存列存储数据库中优化的混合自适应索引

薛忠斌 周 烜 张延松 周 新 王 珊
(教育部数据工程与知识工程重点实验室(中国人民大学) 北京 100872)
(中国人民大学信息学院 北京 100872)

摘 要 分析型数据库在现代企业中得到广泛应用,在使用过程中对查询处理速度的要求逐渐提高。大数据环境下,分析型数据库面临一系列新的挑战:首先,数据复杂性与日俱增,使得数据库系统的初始配置任务更加繁重,例如索引创建等;其次,在分析过程中,由于查询负载模式无法预知,需要对某些属性反复构建索引,以满足查询的时间要求。显然,传统的索引构建维护技术不能完全满足新的应用环境。数据库分裂技术提出了一种不同的策略去解决这些问题。使用数据库分裂技术,DBA不需要对数据库进行细粒度的系统配置。在查询执行过程中,数据库能自动调整以适应查询负载;随着查询负载的变化,系统自动调整索引。近年来,一系列数据库分裂算法被提出,但已有的算法都各有优缺点。因此给出了一个cache conscious的数据库分裂代价模型,并基于该模型构建了一个新的自适应索引,其可以综合不同数据库分裂算法的优势。通过大量实验验证了这种新自适应索引技术的有效性。

关键词 自适应合并,数据库分裂,自适应索引,混合算法
中图法分类号 TP301 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2015.11.004

Optimized Adaptive Hybrid Indexing for In-memory Column Stores

XUE Zhong-bin ZHOU Xuan ZHANG Yan-song ZHOU Xin WANG Shan
(MOE Key Laboratory of Data Engineering and Knowledge Engineering, Renmin University of China, Beijing 100872, China)
(School of Information, Renmin University of China, Beijing 100872, China)

Abstract Analytical database has been widely deployed in modern corporations which are posing increasing demand for the speed of data analysis. In the era of big data, analytical database is faced with a number of new challenges. Firstly, the complexity of data keeps increasing, therefore, more efforts have to be put into system configuration, such as index creation. Secondly, without prior knowledge about the patterns of workload, system administrators have to build and rebuild indexes repeatedly, in order to meet the time constraints. Apparently, traditional approaches to index construction and maintenance can not work well in the new environment. Database cracking provides an alternative to solve the problem. Using database cracking, a DBA does not need to fine-tune the system configuration. Instead, the database can automatically adjust itself to fit the workload during query execution. In recent years, a series of database cracking algorithms have been proposed, while none of them is optimal in all situations. The paper proposed a cache conscious cost model for database cracking. Based on the model, we created a new adaptive index, which can combine the advantages of several previous cracking approaches. Extensive experiments were conducted to demonstrate the effectiveness of our method.

Keywords Adaptive merging, Database cracking, Adaptive indexing, Hybrid algorithm

1 应用背景

现代商业应用中不断产生大量的数据,通过对数据进行分析挖掘来获取其内在价值,成为企业关注的焦点。分析型数据库系统作为数据挖掘的重要工具,得到了学术界和数据库厂商的广泛关注,并产生了一系列的产品,例如荷兰阿姆斯特丹大学设计的MonetDB数据库系统、SAP的HANA数据处理平台、Oracle的内存数据库和IBM的DB2等分析型数据库系统。在分析型数据库系统中列存储和内存分析成为主流的技术模式。

在大数据环境下,分析型数据库具有一些富有挑战性的问题:数据预处理过程中创建索引需要耗费大量时间;工作模式不能预先确定,在分析执行过程中需要对某些属性构建索引,以满足查询响应时间的要求;工作负载动态变化,当数据库中追加新的数据后,需要对索引进行重构。传统的分析型数据库系统不能很好地应对这些问题,于是数据库分裂技术

到稿日期:2014-10-24 返修日期:2014-12-15
薛忠斌(1985—),男,博士生,主要研究领域为内存数据库、移动对象数据库;**周 烜**(1979—),男,博士,副教授,主要研究领域为数据库;**张延松**(1973—),男,博士,讲师,主要研究领域为内存数据库、数据仓库、OLAP;**周 新**(1987—),女,博士生,主要研究领域为时空数据库;**王 珊**(1944—),女,硕士,教授,主要研究领域为高性能数据库新技术、内存数据库新技术、Video数据库技术、数据仓库、大数据管理与分析技术等。

(Database Cracking)[5-7,10,12-15]被提出，并成为近期研究的热点。数据库分裂技术主要应用于内存列存储数据库系统中，具有以下特点：数据预处理过程中不需要创建索引；在查询执行过程中依据工作负载动态构建索引；索引部分数据并在查询过程中进行完善。

数据库分裂技术结合自适应合并技术(Adaptive Merging)[3,4]产生了一系列混合算法[8]。两种有代表性的算法为HCC算法和HCS算法。在负载动态变化时，两种算法各有优劣。通过图1可以看到，HCC算法在执行前期查询响应速度快，随着查询的不断执行，索引进行自适应地创建，在执行后期，HCS算法的查询响应速度快。

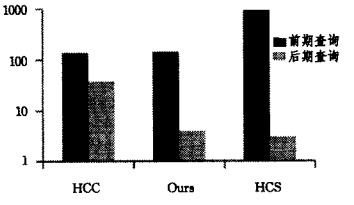

图1　3种算法响应时间对比

本文首先给出了一个cache conscious的代价模型，并基于代价模型，提出了混合索引算法HCCS。HCCS算法同时具有HCC算法前期查询响应速度快和HCS算法后期查询响应速度快的特性。通过实验验证，混合索引在不同的工作场景下取得了很好的性能表现。

本文的创新点有以下3个方面：

1)给出了一个cache conscious的定量的代价模型。在内存算法中，cache miss对算法的性能影响很大。已有的crack算法没有量化的代价模型，无法根据负载的变化对算法进行优化选择。结合硬件的参数，提出了一个cache conscious的代价模型。在构建代价模型过程中，发现多种硬件参数会对程序产生影响，为了便于算法向数据库中移植，选取了对程序执行的结果有决定性影响的参数。

2)提出一种新的算法。基于量化的代价模型，给出了一种新的混合索引算法，新算法结合了已有算法的优势，同时能够根据负载的变化进行动态调整，使查询保持较快的响应速度。

3)在不同的查询工作负载下，对算法进行测试。算法用C++实现，采用了文献[8]中的实验设置，并在随机分布、扭曲分布等数据集下进行了测试，实验显示提出的算法有很好的性能表现。

2　相关工作

文献[10]首先提出了数据库分裂技术(Database Cracking)，通过执行SQL语句模拟实现算法，定性地分析了算法的优势。在文献[5]中数据库分裂技术被嵌入到分析型数据库MonetDB中，算法主要应用于主存列存储模式。

随着对数据库分裂技术研究的不断展开，文献[6]提出了数据库分裂技术相关的更新技术。多列索引技术、支持选择、元组的重构和复杂查询等在文献[7]中提及。结合划分、合并操作的数据库分裂技术出现在文献[3,4,8]中。文献[9]解决了数据库分裂技术在不同的查询场景下的健壮性问题。文献[12]对比了当前已有的数据库分裂相关算法，指出数据库分裂算法进一步优化的方向，同时对于算法选择具有一定的指导意义。文献[14,15]分别介绍了自适应索引在多核的环境下和事务处理中的应用。

自适应合并技术(Adaptive Merging)通过一种增量合并排序(Merge Sort)的方式实现，在一次查询中把多个数据块中的数据合并到最终数据块中。相对于数据库分裂技术，自适应合并技术经过更少的查询便可以完成数据的排序。

3　优化的混合自适应索引技术

结合数据库分裂技术和自适应合并技术，一系列混合自适应索引技术被提出[8]。文献[8]中实验显示，Hybrid Crack Crack(HCC)和Hybrid Crack Sort(HCS)比较有代表性。为了结合HCC算法和HCS算法的优势，提出了一个cache conscious的定量的代价模型。在内存算法中，cache conscious是一个重要的优化思想，是解决内存访问延迟的重要技术手段，但已有的自适应索引算法中没有相关研究。基于代价模型可以量化执行过程中自适应索引的构建和查询的执行情况，根据负载的变化进行查询优化。

代价模型主要是对算法的操作进行分析，基于功效函数$utility=F(T)$进行验证，其中T为查询的响应时间。

在HCC算法和HCS算法的执行过程中，对数据块进行两种操作：分裂(Crack)操作和排序(Sort)操作。对数据块进行分裂操作时，关键字的比较次数为$O(n)$；执行排序操作时，关键字比较次数为$O(n\lg n)$。只能通过扫描的方式对分裂操作产生的数据块进行查询，关键字的比较次数为$O(n)$；对有序的数据块可以进行折半查找(Binary Search)，关键字的比较次数为$O(\lg n)$。

对数据块进行分裂操作时，在当前查询中获得的收益为$F(TS)-F(TC)$。对数据块进行排序操作时，后续查询获得的收益为$F(TC)-F(TB)$。

当$F(TC)-F(TB)>F(TS)-F(TC)$时，即当后续查询通过折半查找获得的时间收益大于当前查询排序付出的代价时，对数据块进行排序操作。通过分析发现：在执行过程中数据块的大小及系统中Cache Size、TLB Entries、Bandwidth等硬件参数会对程序产生影响。经过进一步分析发现，Cache Size作为一个重要的指标，对程序执行的结果有决定性的影响。因此以Cache Size作为代价模型的替代指标。当分裂后的数据块中包含数据的大小满足Cache Size时进行排序操作。通过选择简单的指标，方便算法的跨平台操作，同时易于向数据库中移植。

基于代价模型，提出了一种新的混合算法——HCCS(Hybrid Crack Crack Sort)，算法的伪代码如下。

算法1　自适应范围查询Hybrid Crack Crack Sort算法

输入：范围查询的边界值a,b，初始数据块IP，最终数据块FP，阈值θ

输出：查询结果

```
Hybrid Crack Crack Sort(a,b,IP,FP,θ)
1. Search in FP
2. FP1=the Piece of FP that contains value a
3. FP2=the Piece of FP that contains value b
4. result1=Search_FP(FP1,a)
5. result1=result1 union Search_FP(FP2,b)
6. if (result1 not full answer the query)
7.     Search in IP
8.     for each partition in IP
9.         IP1=the Piece of IP that contains value a
10.        IP2=the Piece of IP that contains value b
```

```
11.        Temp_Partition=Temp_Partition union Crack(IP1,a)
12.        Temp_Partition=Temp_Partition union Crack(IP2,b)
13.    If(the sizeof Temp_Partition < θ)
14.      Sort(Temp_Partition)
15.    Merge Temp_Partition into FP
16. return result1 union Temp_Partition
```

对于到来的查询，HCCS算法首先在最终块中进行查找，如算法中的1－5行，当得到的结果不能完全满足查询要求时，再在初始数据块中进行查询。当分裂后数据块满足阈值时，继续执行排序操作。最后把得到的结果和初始数据块中的数据合并到最终数据块中，从而完成查询和索引的自适应构建。

通过一个实例来解释算法。

图2为混合分裂分裂排序(HCCS)算法和混合分裂排序(HCS)算法的对比实例。在算法中首先将原始数据划分到各个数据块中，数值在数据块内无序。对于HCCS算法，为了简便，用数据块中数值项数目作为阈值，设定为3。对于HCCS算法，在执行第一个查询时，根据代价模型，从初始数据块中得到的结果不执行排序操作，直接合并到最终数据块中，与HCC算法相同。对于最终数据块中分裂的数据块和保存查询结果的临时数据块，当满足阈值条件时，执行排序操作，与HCS算法相同。通过上面的例子可以看出，HCCS算法同时具有HCC算法前期查询响应速度快和HCS算法数据收敛速度快的优势。同时相对于HCC算法和HCS算法，HCCS算法更灵活，可以根据用户的需要调整阈值。通过后续的实验发现，当阈值设置为L3 cache大小时，系统的整体响应时间较好。所以在默认的情况下，设置L3 cache大小作为阈值。

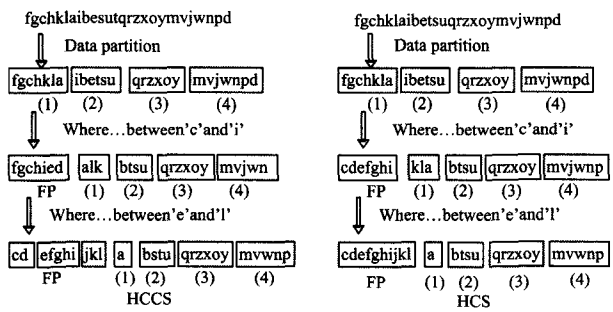

图2 混合分裂分裂排序算法与混合分裂排序算法对比

4 实验分析

本节通过实验来验证HCCS算法相对于HCC算法和HCS算法更具有优势。

实验设计：实验中的算法用C++实现，用STL库(Standard Template Library)实现分裂索引。所有的实验在12核(2 Intel(R) Xeon(R)CPU E5645@2.40GHz)服务器上实现，服务器有48GB RAM，12MB L3 Cache，操作系统为Red Hat Enterprise Linux Server release 5.5 (Tikanga)。实验参照文献[8]中的实验设计。实验在主存中进行，用于模拟主存列存储系统，执行的查询模式和数据与文献[8]相同。查询模式为：

select A from R where A>low and A<high

实验过程中使用一个包含3×10^8个元组的列，列中的值在[0,10^8)中随机分布。查询执行过程中，使用文献[9]中部分查询模式，如图3所示。

在执行测试时，在3种查询负载下连续进行了1000个选择率为10%的查询，用于完成索引的自适应构建。为了进一步显示在查询初期3种算法性能的差异，在结果中对坐标轴采用对数(log)方式进行表示。同时在结果中用Scan和Sort查询执行的结果作为对照。

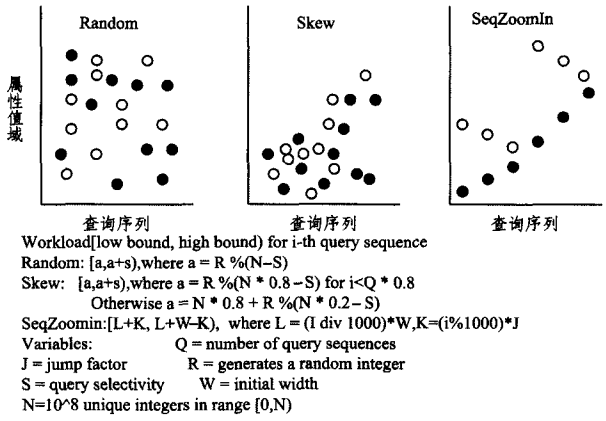

图3 查询工作负载模式

图3示出了3种常用的查询工作负载模式，分别为随机查询负载模式、查询扭曲时的负载模式和查询缩进的负载模式。其中查询缩进的负载模式中，查找的范围会逐渐减小，这种查询模式对当前算法影响较大，通过后面的实验可以发现，HCCS算法在这种情况下的性能与表现最好的算法的性能相同。

在不同的阈值下对算法进行了测试，如图4所示。在前期查询中，当阈值为16MB时，查询的响应时间较长。当阈值为8MB时，在查询后期，数据没有完成收敛，大部分查询响应时间较长。通过进一步统计查询总的响应时间，可以发现当阈值设置为12MB时，查询总体响应时间最好。即当用户关注于查询前期响应时间时，可以调小阈值，使前期查询产生的数据块只执行分裂操作，从而缩短查询的响应时间。当用户关注于数据的收敛速度时，可以适当调大阈值，索引的排序操作尽量在前期查询时完成，使数据能够快速地收敛。但从总体的响应时间发现，当阈值设置为L3 cache size时，查询的总体响应时间最优。在后面的实验中设置阈值为12MB，即L3 cache的大小。

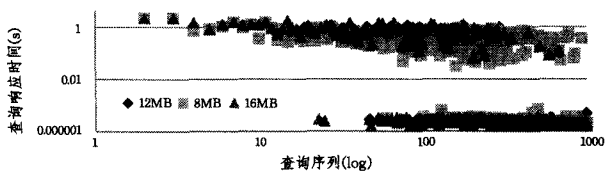

图4 HCCS在不同查询阈值下的性能表现

4.1 随机查询工作负载

4.1.1 快速的查询响应速度

对HCC、HCS和HCCS 3个算法在随机查询负载下(如改变select A from R where A>low and A<high查询中的low和high值)在选择率为10%时执行1000个不同的查询。用Scan和Sort查询结果作为对照。对比图5和图7，在查询初始阶段，HCCS算法和HCC算法的查询结果相近，部分结果相同。对比图5－图7，在前10个查询中，HCS算法的查询响应时间最长，有7个查询的响应时间要大于对应扫描查询的响应时间。在HCS算法中，从初始数据块中查到的结果

需要经过排序然后才放入最终数据块中，排序操作耗费大量的时间。HCC 算法的查询响应时间最短，前 10 个查询的结果都小于对应扫描方式的结果。在 HCC 算法中，只需把查询结果放入最终数据块中，没有额外操作。HCCS 算法中用代价模型来决定是否对数据块进行排序操作。在查询初期，HCCS 算法比 HCS 算法快，与 HCC 算法性能接近。HCCS 算法中，只有两个查询的结果大于对应扫描方式的结果。

对于第 9 个查询，HCS 算法执行结果优于 HCC 算法和 HCCS 算法，与对应 Sort 查询执行的结果相同。在 HCS 算法中，第 9 个查询涉及到的数据在前面查询中已经有序。查询执行时，在最终数据块中通过二分法进行查找。对于 HCC 算法，必须扫描查询值对应的数据块。在 HCCS 算法中，基于代价模型，查询涉及的数据块没有被排序，需要通过分裂的方式进行查询。

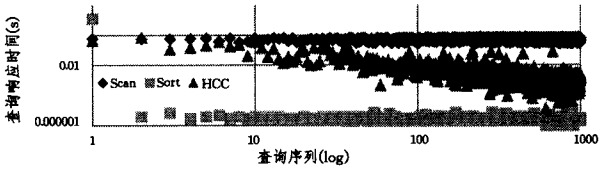

图 5　随机查询负载下 10%选择率的 HCC 算法结果

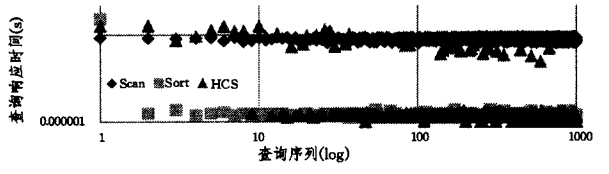

图 6　随机查询负载下 10%选择率的 HCS 算法结果

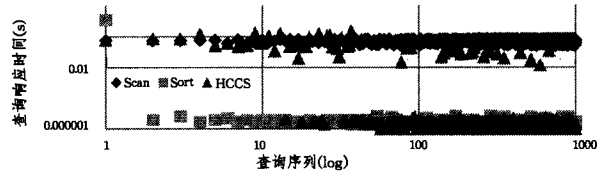

图 7　随机查询负载下 10%选择率的 HCCS 算法结果

4.1.2　快速的数据收敛速度

图 6 中，在 10%选择率下，HCS 算法能快速完成数据收敛。在 HCS 算法中，先对从初始数据块中查到的结果进行排序，然后放入最终数据块中。当选择率在 10%时，经过十几个查询后，大部分数据已经有序。当后续查询涉及到有序数据块时，通过折半法进行查找，其查询的响应时间和排序后查询的响应时间相同。图 5 中，HCC 算法的查询响应时间为一个平滑的曲线。在 HCC 算法中，通过分裂方法进行查询，每次查询都需要扫描查询值对应的数据块，查询过程中数据没有完成收敛，查询响应时间始终大于排序后查询的响应时间。对于 HCCS 算法，通过图 7 可以看到，在查询前期，查询响应速度较快。在查询前期得到的数据块不满足设定的阈值，基于代价模型，不需要对数据块进行排序操作，从而确保了前期查询响应时间。随着查询的不断执行，当临时数据块和最终数据块中分裂的数据块满足代价模型时，执行排序操作，加快数据的收敛速度。当后续查询涉及到有序数据块时，通过折半法进行查找。图 7 显示，HCCS 算法完成数据收敛时，查询的响应时间与排序后查询的响应时间相同。

4.2　查询扭曲工作负载

在 Skew 查询负载下进行了进一步的测试。在图 3 的查询工作负载模式图中可以看到，Skew 查询负载中查询明显集中于某几个区域。

4.2.1　均衡的总体查询响应时间

对比图 8 和图 9 可以发现在数据倾斜查询负载下，在查询初期 HCC 算法的查询响应速度较快，前 10 个查询的响应时间都小于对应扫描方式的响应时间；在 HCS 算法中，有 7 个查询的响应时间接近或大于对应扫描方式的查询响应时间，有 3 个查询的响应时间与排序后查询的响应时间相近。对于 HCCS 算法，在前 10 个查询中，第 3 个和第 4 个查询的响应时间与 HCC 算法中对应查询的响应时间相同，第 7 个、第 8 个和第 9 个查询的响应时间与 HCS 算法中对应查询的响应时间相同。同时通过图 10 可以看到 HCCS 算法已经完成数据收敛，所以其性能表现优于 HCC 算法和 HCS 算法。代价模型使得 HCCS 算法同时具有 HCC 算法的前期查询响应快和 HCS 算法的数据收敛快的特性。

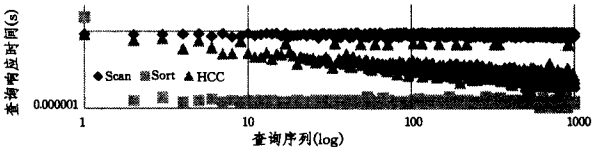

图 8　扭曲查询负载下 10%选择率的 HCC 算法结果

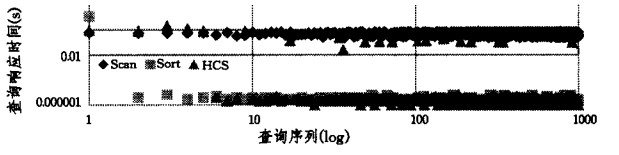

图 9　扭曲查询负载下 10%选择率的 HCS 算法结果

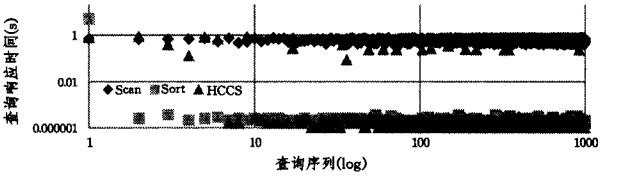

图 10　扭曲查询负载下 10%选择率的 HCCS 算法结果

4.3　查询缩进工作负载

通过图 3 查询工作负载模式图可以看到，SeqZoomIn 查询负载为一个分组式的查询模型。对于每个查询分组，组内的第一个查询首先查找一个大的数据范围，然后逐渐减少查询的范围。每组查询集中于一个数据区域，对区域内的值进行层层细化查询。在这种查询模式下，第一个查询会把本组查询涉及的所有数据项放入最终数据块中，小组中的后续查询只需要在最终数据块中查找便可完成查询。

在当前查询模式下，数据能很快完成收敛，所以不需要通过扫描方式与排序后查询结果进行对比，直接对这 3 种算法进行比较。通过图 11 的最高点值可以看到，共执行了两组查询。对于第一组查询，其数据已经完全在 Cache 中，不需要替换，同时数据量不是很大，通过分裂方法的查询时间与折半查找的查询时间相差不大。当第二组查询到来时，需要从初始数据块中查找，进行 Cache 替换，所以其查询响应时间较长。同时可以看到，在这种查询模式下 HCS 算法优于 HCC 算法。HCCS 算法的性能与 HCS 算法相同。

（下转第 36 页）

化，相对于优化前版本，整体取得了 1.8～13.3 倍的性能提升。但是由于时间所限，还有很多工作需要完善，如算法的指令级优化、访存锁优化等。由于 NVIDIA 690 的 OpenCL 原子操作是由软件实现的，导致本算法的实现未能在 NVIDIA 690 上获得理想的性能，之后也将对此做进一步研究。

参 考 文 献

[1] Jia Hai-peng. Research of Parallel Optimization Technicals on GPU Computing Platforms[D]. Qingdao: Ocean University of China, 2013

[2] Shame R, Kennedy R A. Efficient histogram algorithms for NVIDIA CUDA compatibledevice [C] // ICSPCS2007. New York: IEEE, 2007: 418-422

[3] Di Peng, Hu Chang-jun, Li Jian-jiang. Efficient Method for Histogram Generetionon GPU[D]. Beijing: University of Science and Technology, 2011

[4] Gómez-Luna J, González-Linares J M, Benavides J I, et al. An optimized approach to histogram computation on GPU[J]. Machine vision and applications, 2013, 24(5): 899-908

[5] Zhang Yuan-quan, ZhangXian-yi, Jia Hai-peng, et al. Heterogeneous Computing with OpenCL [M]. Tsinghua University press, 2012

[6] AMD GRAPHICS CORES NEXT(GCN) Architecture Whitepaper [J/OL]. https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf

[7] Munshi A, Gaster B, Mattson T G, et al. OpenCL programming guide[M]. Pearson Education, 2011

[8] AMD R & D center in Shanghai. Cross platform multicore and manycore Programming Notes——int the way of OpenCL[OL]. http://down.51cto.com/data/964762

[9] AMD. AMD Accelerated Parallel Processing OpenCLTM Programing Guide [OL]. http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf

[10] Zhang Jing. OpenCV2 Computer vision programming manual [M]. Science Press Limited liability company, 2013

[11] Jia Hai-peng, Zhang Yun-quan, Long Guo-ping, et al. GPURoofline: A Model for Guiding Performance Optimizations on GPUs [C]// Proceeding of International European Conference on Para-llel and Distributed Computing. Rhodes Island, Greece, 2012: 920-932

[12] Jia H, Zhang Y, Wang W, et al. Accelerating viola-jones facce detection algorithm on gpus[C]// 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS). IEEE, 2012: 396-403

[13] NVIDAI. GPU-ACCELERATED APPLICATIONS [OL]. http://www.nvidia.com/object/media-and-entertainment.html

[14] NVIDIA. NVIDIA's Next Generation CUDA™ Compute Achitecture: Kepler GK110 [OL]. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

(上接第 31 页)

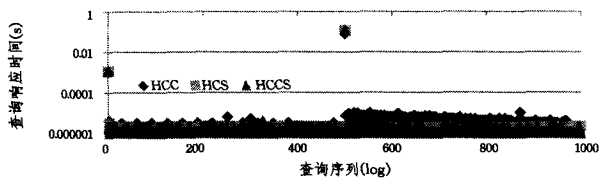

图 11 SeqZoomIn 查询负载下的结果

结束语 在海量数据下，分析型数据库系统面临索引构建和维护、查询效率等一系列新的挑战。为了应对挑战，数据库分裂技术、自适应合并技术及一系列的混合算法被提出并成为近期关注的热点。

本文在对已有算法分析的基础上，给出了一个定量的代价模型。已有的研究没有量化的代价模型，导致对于查询执行过程中自适应索引的构建和查询的响应无法进行量化分析，难以对动态负载下的算法进行优化选择。基于代价模型，我们给出一个新的自适应索引算法。新的算法更灵活，同时具备已有算法的优点。在不同的查询工作负载下的实验表明，新的混合算法具有很好的性能表现。

参 考 文 献

[1] Chaudhuri S, Weikum G. Rethinking database system architecture: Towards a self-tuning risc-style database system [C] // Proceedings of the 26th In VLDB. 2000: 1-10

[2] Graefe G, Idreos S, Kuno H, et al. Benchmarking adaptive indexing[C]// TPCTC. 2010: 169-184

[3] Graefe G, Kuno H. Adaptive indexing for relational keys[C]// ICDE. 2010: 69-74

[4] Graefe G, Kuno H. Self-selecting, self-tuning, incrementally optimized indexex[C]// EDBT. 2010: 371-381

[5] Idreos S, Kersten M L, Manegold S. Database cracking[C]// CIDR. 2007: 68-78

[6] Idreos S, Kersten M L, Manegold S. Updating a cracked database[C]// SIGMOD. 2007: 413-424

[7] Idreos S, Kersten M L, Manegold S. Self-organizing tuple reconstruction in column stores[C]// SIGMOD. 2009: 297-308

[8] Idreos S, Manegold S, Kuno H, et al. Merging what's cracked, cracking what's merged: adaptive index in main-memory column-stores[J]. PVLDB, 2011, 4(9): 585-597

[9] Halim F, Idreos S, Karras P, et al. Stochastic database cracking: Towards robust adaptive indexing in main-memory Column-stores[J]. PVLDB, 2012, 5(6): 502-513

[10] Kersten M, Manegold S. cracking the database store[C]// CIDR. 2005: 213-224

[11] Hoare C A R. Algorithm 64: Quicksort[J]. Communications of the ACM, 1961, 4(4): 319-320

[12] Schuhknecht F M, Jindal A, Dittrich J. The Uncracked Pieces in Database Cracking[C]// VLDB. 2013

[13] Pirk H, Petraki E, Idreos S, et al. Database Cracking: Fancy Scan, not Poor Man's Sort! [J/OL]. http://oai.cwi.nl/oai/asset/22474/22474D.pdf

[14] Graefe G, Halim F, Idreos S, et al. Transactional support for adaptive indexing[J]. The VLDB Journal, 2014, 23(2): 303-328

[15] Alvarez V, Schuhknecht F M, Dittrich J, et al. Main Memory Adaptive Indexing for Multi-core Systems[J/OL]. http://arxiv.org/abs/1404.2034