计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2016.20151163
53(4):798-810, 2016

基于代价估计的 Hive 多维索引分割策略选择算法

刘 越[1,2] 李锦涛[1] 虎嵩林[3]
[1](中国科学院计算技术研究所 北京 100190)
[2](中国科学院大学 北京 100049)
[3](中国科学院信息工程研究所 北京 100093)
(liuyue01@ict.ac.cn)

A Cost-Based Splitting Policy Search Algorithm for Hive Multi-Dimensional Index

Liu Yue[1,2], Li Jintao[1], and Hu Songlin[3]
[1](*Institute of Computing Technology, Chinese Academy of Sciences, Beijing* 100190)
[2](*University of Chinese Academy of Sciences, Beijing* 100049)
[3](*Institute of Information Engineering, Chinese Academy of Sciences, Beijing* 100093)

Abstract In the domain of energy Internet, smart city, etc, the massive smart devices collect large amount of data every day, and traditional enterprises need to perform lots of multi-dimensional analysis on these data to support decision-making. Recently, these enterprises try to solve the big data problem with technologies from Internet companies, for example, Hadoop and Hive etc. However, Hive has limited multi-dimensional index ability, and cannot satisfy the requirements of high-performance analysis in traditional enterprises. In this paper, we propose a distributed grid file based multi-dimensional index—DGFIndex to improve the multi-dimensional query performance of Hive. However, DGFIndex needs user to specify the splitting policy when creating index, which is not trivial for user when they are not familiar with data and query pattern. To solve it, we propose a novel MapReduce cost model to measure the DGFIndex-based query performance on specific splitting policy, a two-phase simulated annealing algorithm to search for the suitable splitting policy for DGFIndex, and finally decrease the total cost time of query set. The experimental results show that, DGFIndex improves 50%~114% query performance than original Compact Index in Hive. For static query set, compared with manual-specifying partition policy, our algorithm can choose suitable interval size for each index dimension, and decrease the cost time of query set at most 30%.

Key words Hive; MapReduce; multi-dimensional index; cost model; simulated annealing

摘 要 在能源互联网、智慧城市等新兴领域,智能终端采集的庞大数据往往需要多维分析,传统企业寻求借助互联网技术(如 Hadoop 和 Hive)应对大数据问题.但是 Hive 当前的多维索引能力较弱,无法满足传统企业的需求.针对这一问题,提出了一种基于分布式网格文件的多维索引技术——DGFIndex,来提升 Hive 的多维查询处理能力.但是在创建 DGFIndex 时,需要用户指定各个索引维度的分割粒度,而分割粒度的大小与查询性能息息相关.在用户对数据与查询特征不熟悉时,很难选择较优的分割策略.为了解决这一问题,通过建立新的 MapReduce 代价模型,并使用两阶段模拟退火算法为 DGFIndex 搜

收稿日期:2015-12-21;**修回日期**:2016-02-02
基金项目:国家自然科学基金项目(61070027)
This work was supported by the National Natural Science Foundation of China (61070027).
通信作者:虎嵩林(husonglin@iie.ac.cn)

索较优的分割策略,从而提升查询性能,减少查询集合的总耗时.实验结果表明:DGFIndex 可以提升 Hive 多维查询性能 50%～114%,对于固定的查询集合,与人工选定分割策略比较,基于代价估计的分割策略选择算法可以为 DGFIndex 快速选定较优的分割策略,并可以使整个查询集合的处理时间比人工方法最多减少 30%.

关键词　Hive;MapReduce;多维索引;代价模型;模拟退火

中图法分类号　TP311.132

随着能源互联网、智慧城市等领域技术的发展,传统企业,如电力、通信、金融、制造等行业,需要处理的数据呈现爆炸式增长,以往依靠传统商业关系型数据库的解决方案遇到了瓶颈,其写入吞吐、横向扩展性与大数据分析能力都无法满足日益增长的数据存储与分析需求.因此,传统企业都在寻找应对大数据挑战的替代方案.而互联网领域已经出现了大量的大数据处理系统,如 Hadoop,Spark[1],Storm 等,但是由于互联网应用与传统企业应用各具特点,因此在将互联网技术应用于传统企业时会遇到诸多挑战[2-4].例如,传统企业往往需要对庞大的采集类数据进行快速的在线分析与离线批量计算,在这些计算逻辑中包含大量的多维查询[3],但是现有互联网领域的数据处理系统索引能力有限,无法满足需求.

具备灵活可扩展性、高可用性、良好容错性与低成本等优点的 Hadoop 已成为海量数据存储与分析的首选方案之一.而 Hadoop 之上的数据仓库工具 Hive[5]为用户提供了类 SQL 查询语言 HiveQL,这使传统企业的数据分析人员可以平滑地过渡到基于 Hadoop 的数据平台上.但是传统企业的数据分析逻辑包含大量的多维查询与统计值分析,而 Hive 只支持有限的索引技术,如 Compact Index 等,这些索引数据过滤粒度较大,会造成过多的冗余数据读取,查询性能较低.为此,我们提出了一种基于分布式网格文件的多维索引技术——DGFIndex(distributed grid file index)[3].DGFIndex 使用网格文件将数据空间分为很多小单元,通过小单元坐标与对应数据片位置的映射关系达到细粒度索引的目的.

在创建 DGFIndex 时,用户需要指定每个索引维度的最小值与分割区间,但是在不熟悉数据与查询特征时,选择较优的分割策略比较困难.如果索引维度分区粒度过小,虽然索引可以更精确地定位查询相关数据,即减少冗余数据读取,但是数据读取性能较低;如果索引维度分区粒度过大,虽然数据读取性能较高[6],但是会造成读取过多的冗余数据,并且数据解析与数据过滤的 CPU 开销也会增大.因此,自动地为 DGFIndex 选择较优的分割策略对查询性能的提升至关重要,但是不同查询间的最优分割策略会相互影响与制约.在传统企业中,多数数据分析逻辑都以存储过程形式存在,即查询集合在一定时间内是静态的,所以本文要解决的问题是:对于固定的查询集合,如何为 DGFIndex 选择最优的分割策略,从而使查询集合耗时最短.为了解决该问题,本文首先建立了一种新的 MapReduce 代价模型来评估不同分割策略对查询开销的影响,然后基于该代价模型提出了一种基于模拟退火的两阶段搜索算法加速分割策略的搜索过程,最终为查询集合选择较优的 DGFIndex 分割策略,减小查询集合的总耗时.本文的主要贡献将是:

1) 根据采集类数据的特征与查询特点,提出了一种面向 Hive 的分布式网格文件索引——DGFIndex.该索引可以减少查询冗余数据读取量,提升查询性能.

2) 针对基于索引查询的 MapReduce 处理过程的特征,提出了一种新的 MapReduce 代价模型,并基于此提出了一种基于模拟退火的两阶段分割策略搜索算法.

3) 在实验中,本文使用不同的查询集合大小、数据集大小评测分割策略搜索算法的有效性,实验显示该算法可以快速地找到较优的分割策略,从而减少查询集合的总耗时.

1　相关工作

在 Hadoop 索引研究领域,现有的研究工作主要分为 2 类:1)应用于传统数据分析的索引研究,如 Jiang 等人[7]提出了一种 HDFS 上的基于排序文件的一维区间索引,该索引为每个固定大小的页建立一个索引项,索引表记录该页的最小值、最大值以及偏移量,而页的大小由用户指定.Dittrich 等人[8]提出了 Trojan Index 和 Trojan Join Index,前者为每个

数据片保存最小值、最大值和记录数,该数据片的大小由用户指定,后者通过将欲连接的数据集按照连接维度共同分区存储到一起来加速多表连接操作. Eltabakh 等人[9]提出了 Eagle-eyed elephant 系统,该系统提供了多种索引机制来加速数据读取,包括为每个数据片内的数值和日期类型建立区间索引、为字符串类型建立倒排索引等,该数据片的大小为 HDFS 块大小. Richter 等人[10]提出了 HAIL,该系统提供了一种静态索引与一种自适应索引,该索引为每个数据块副本建立不同的索引结构,在该研究工作中索引数据片的大小是固定的. 2)应用于空间数据处理的索引研究,如 Aji 等人[11]提出的 Hadoop GIS,该系统提出了一种 2 层的索引结构,全局索引保存维度值与对应数据片的映射关系,在每个节点为每个数据片根据查询类型按需建立局部索引. 此外,Eldawy 等人提出了 Spatial Hadoop[12],该系统也提供了一种 2 层索引结构,首先使用 Grid File, R-Tree 或 R+-Tree 将数据划分为大小相同的块,然后为每个块建立局部索引,全局索引保存在主节点的内存中以加速索引的访问. 从上面的描述可以看出现有的 Hadoop 索引相关的研究工作,索引粒度或者由用户指定或者设定为固定经验值,并没有考虑查询集合的影响.

在 Hadoop 代价模型领域,Herodotou[13]为 Hadoop 的 MapReduce 任务执行流程的各个阶段提出了详细的代价模型,该模型应用于 Hadoop 最优化参数选择[14]. Wang 等人[4]提出了一种面向 Hive 的代价模型,用于多表连接最优顺序选择. Lin 等人[15]为 MapReduce 提出了一种向量形式的代价模型,并基于该代价模型估算任务耗时. Wang 等人[16]针对偏斜数据的 GroupBy 查询和 Join 查询提出了估价模型. Song 等人[17]针对 MapReduce 中二元连接查询提出了 I/O 代价模型. 但是以上工作没有考虑基于索引的情况,即其 Map 阶段的数据读取直接使用本地顺序读取或网络读取性能指标进行代价估计,但是我们发现数据读取性能不仅与数据片大小有关,还与读取的查询相关数据量有关. 此外,每个 Mapper 读取的数据量不再等于 HDFS 块大小,而是经过索引定位后的数据量. 并且,如果查询处理需要多遍 Map 阶段时,已有的工作往往使用一遍 Map 阶段的耗时与遍数的乘积作为估计,但是如果存在索引的话,各个 Mapper 读取的数据量是不同的,读取数据量少的 Mapper 可以快速执行完,后面未执行的 Mapper 可以利用空闲出的资源马上执行. 总之,现有 MapReduce 代价模型无法直接应用于基于索引的 MapReduce 处理代价估计中.

2 DGFIndex

2.1 DGFIndex 结构

图 1 展示了一个建立在索引维度 x 和 y 上的 2 维 DGFIndex. DGFIndex 使用网格文件将数据空间分为很多小单元,这些小单元称为 GFU(grid file unit). 这样,表中的记录会按照索引维度的值落在对应的 GFU 中,而所有落在同一个 GFU 中的记录会被连续地存储在数据文件中,称为数据片. 每个 GFU 在索引表中以 Key/Value 的形式存储,*GFUKey* 为 GFU 在数据空间中的左下角坐标,*GFUValue* 由 2 部分组成:*header* 和 *location*. *header* 为用户定义的预计算用户自定义函数(user-defined function, UDF),例如可以预计算每个 GFU 中数值列的 *sum*,这样可以极大地加速聚集值查询;*location* 记录该 GFU 对应数据片所在的文件名、开始偏移量和结束偏移量. 例如,图 1 中 GFU 对应的 *GFUKey*=10_17,如果在创建索引时预计算维度 z 的 *sum*,则 *header* 为落

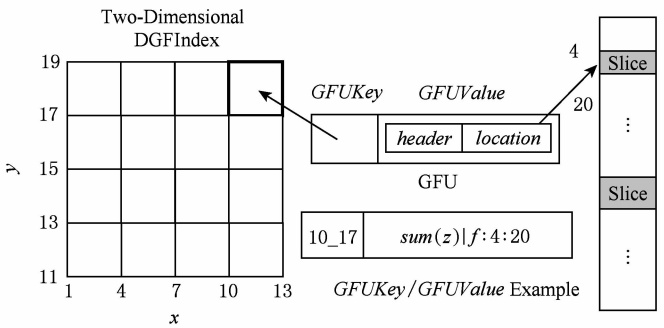


Fig. 1 DGFIndex structure.

图 1 DGFIndex 结构

在该 GFU 中所有记录的 $sum(z)$ 值,*location* 为存储所有落在该 GFU 中记录的数据片的位置信息.在本例中,文件名为 f,开始偏移为 4,结束偏移为 20.数据片的粒度由用户在创建索引时给出的分割策略确定,分割策略包括每个索引维度的最小值和分割区间.如图 1,在创建该索引时,指定索引维度 x 的最小值为 1,分割区间大小为 3;而索引维度 y 的最小值为 11,分割区间大小为 2.此外,为了加快索引表的读取,DGFIndex 的索引表以 Key/Value 的形式保存在 HBase 中,这样可以以基于键值的方式快速读取索引信息.

DGFIndex 可以良好地支持行式存储与列式存储.例如,对于 TextFile 存储格式,DGFIndex 记录每个数据片在文件中的开始偏移量和结束偏移量;对于 RCFile 存储格式,每个数据片保存为其中的一个行组(row group),DGFIndex 只需记录每个 Row Group 的开始偏移量.可以看出,在 DGFIndex 中数据片为最小读取单位.

2.2　DGFIndex 创建索引过程

DGFIndex 的创建过程为一个 MapReduce 任务,Map 阶段的计算逻辑如算法 1 所示,Reduce 阶段的计算逻辑如算法 2 所示.

算法 1. 创建 DGFIndex 的 Map 函数.

输入:键为记录在该数据块中的偏移量,值为记录 *record*;

输出:发送给 Reducer 的 Key/Value 对〈*GFUKey*, *record*〉.

① $idxDValues = getIdxDimValues(record)$;

② $GFUKey = computeGFUKey(idxDValues)$;

③ $Emit(GFUKey, record)$.

在创建索引的 Map 阶段,对于每条记录,首先读取其索引维度的值(行①);然后根据索引维度的值与分割策略计算其所属的 GFU,从而得到该条记录的 *GFUKey*(行②);最后将 *GFUKey* 与该记录作为键值对发往 Reduce 函数(行③).

算法 2. 创建 DGFIndex 的 Reduce 函数.

输入:键为 *GFUKey*,值为具有相同 *GFUKey* 的 *record* 列表 List〈Record〉*recordsList*;

输出:存储在 HBase 中的索引表与重组之后的数据文件.

① $startOffset$ = 当前输出文件的偏移量;

② $endOffset = -1$;

③ $sliceSize = 0$;

④ $filename$ = 当前输出文件的名字;

⑤ $header$ = null;

⑥ for *record* in *recordsList* do

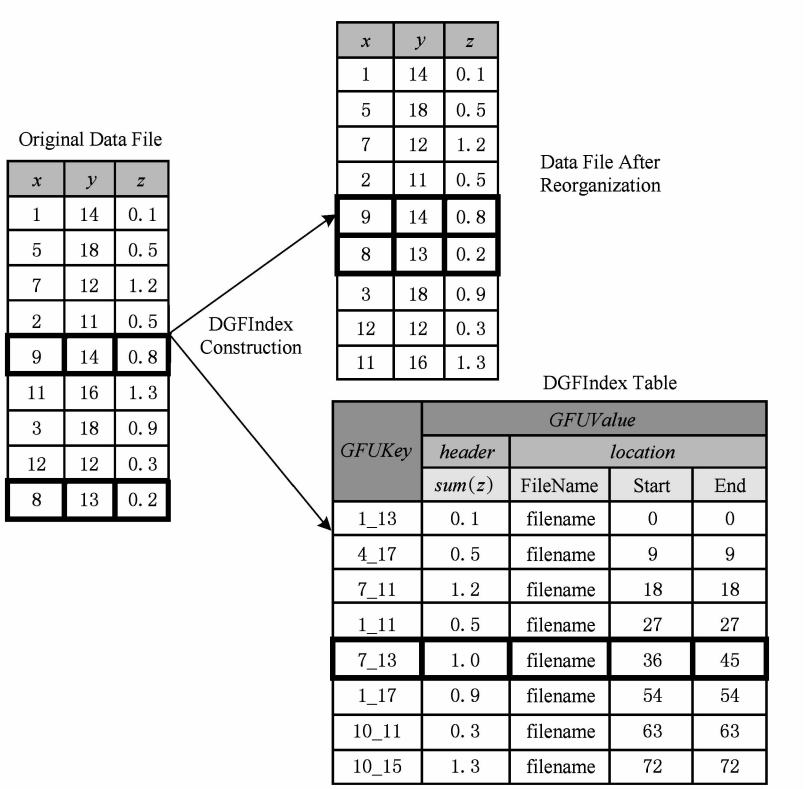


Fig. 2　An example of DGFIndex construction.

图 2　DGFIndex 创建过程例子

⑦ $header=precompute(record,header)$;
⑧ $sliceSize+=sizeof(record)$;
⑨ end for
⑩ $endOffset=startOffset+sliceSize$;
⑪ $GFUValue=\langle header,\langle filename,startOffset,endOffset\rangle\rangle$;
⑫ $KVStore.put(GFUKey,GFUValue)$.

在创建索引的 Reduce 阶段,Reducer 接收到具有相同 *GFUKey* 的记录列表,即属于同一个数据片的所有记录,最终目的是将每个数据片按序输出,并记录 *GFUKey* 与数据片在文件中位置的对应关系.同时,还需按照用户指定的 UDF 预计算每个 GFU 中记录对应的值.具体地,1)初始化数据片开始偏移、结束偏移、数据片大小、输出文件名与保存预计算值的 *header*(行①～⑤);2)对于具有相同 *GFUKey* 的每条记录(行⑥),首先预计算其 UDF 值并累加到 *header* 中(行⑦),然后将当前记录的大小累加到 *sliceSize* 中(行⑧～⑨),待处理完整个记录列表,计算其结束偏移量(行⑩);3)计算 *GFUValue*,并将 *GFUKey*/*GFUValue* 对写入 HBase(行⑪～⑫).至此,索引创建结束.

如图 2 所示,假设现有一个数据文件,使用图 1 中的分割策略创建 DGFIndex,同时预计算 $sum(z)$,可以看到,DGFIndex 索引创建的过程实际为数据重组织的过程,目的为将位于同一 GFU 中的记录存储到一起.如〈9,14,0.8〉与〈8,13,0.2〉位于同一个 GFU,经过数据重组织后存储到数据文件的连续位置.同时,创建了包含 *GFUKey* 与 *GFUValue* 的索引项.

2.3 基于 DGFIndex 的查询处理过程

基于 DGFIndex 的数据查询过程分为 3 步:1)索引表读取;2)HDFS 数据块过滤;3)数据块内部的数据片过滤.下面分别详细描述各步骤.

算法 3. 索引表读取.

输入:SQL 查询 q_i;

输出:查询相关的数据片位置集合 $SLOC_{q_i}$;

查询子结果 *SubRes*,如果查询可以使用预计算的 UDF 时,在这一步可以通过读取索引表得到部分查询结果.

① $idxPred=extract(q_i)$;
② $isUDFQuery=check(q_i)$;
③ $\{imerKeySet,boundaryKeySet\}=DGFIndex.search(idxPred)$;
④ $queryKeySet=boundaryKeySet$;
⑤ if $isUDFQuery$ then
⑥ $SubRes=KVStore.getHeader(innerKeySet)$;
⑦ $writeToTmpFile(SubRes)$;
⑧ else
⑨ $queryKeySet=queryKeySet\cup InnerKeySet$
⑩ end if
⑪ $SLOC_{q_i}=KVStore.getLocation(queryKeySet)$
⑫ $writeToTmpFile(SLOC_{q_i})$.

步骤 1. 索引表读取的过程如算法 3 所示,首先从查询 SQL 中得到索引维度相关的查询谓词 *idxPred*(行①),并判断该 SQL 是否查询预计算 UDF(行②);然后使用查询谓词查询 DGFIndex 得到查询相关的 *GFUKey* 集合(行③).查询相关的 GFU 分为 2 类:1)内部 GFU—*innerKeySet*,即完全位于查询区域内部的 GFU;2)边界 GFU—*boundaryKeySet*,即与查询边界相交的 GFU.内部 GFU 内的数据一定全部为查询相关的数据,但是边界 GFU 内的数据不一定满足查询谓词.然后将需要从 HBase 读取的 *GFUKey* 集合初始化为边界 *GFUKey* 集合 *boundaryKeySet*(行④).如果查询为查询预计算 UDF 的查询,则对于内部 *GFUKey* 集合直接读取 HBase 中预计算的值,得到查询子结果并写入临时文件,待与后面得出的结果合并(行⑤～⑦).如果查询为普通查询,则也需要读取内部 GFU 中的数据,这时将 *InnerKeySet* 也并入需要查询的集合(行⑧～⑩).最后得到查询 q_i 相关的数据片集合的位置信息,并将其写入 HDFS 中的临时文件,待后面数据过滤使用(行⑪～⑫).

算法 4. HDFS 数据块过滤.

输入:输入文件列表 *fileList*、保存 $SLOC_{q_i}$ 的临时文件的路径;

输出:经过索引过滤后的查询 q_i 相关的 HDFS 数据块集合 $chosenBlocks_{q_i}$.

① $chosenBlocks_{q_i}=\varnothing$;
② $SLOC_{q_i}=readFromTmpFile()$;
③ $allBlocks=getSplits(fileList)$;
④ for *block* in *allBlocks* do
⑤ if $block\cap SLOC_{q_i}\neq\varnothing$ then
⑥ $chosenBlocks_{q_i}=chosenBlocks_{q_i}\cup block$;
⑦ end if
⑧ end for
⑨ for *block* in $chosenBlocks_{q_i}$ do
⑩ $slicesInBlock=getRelatedSlices(SLOC_{q_i})$;

⑪　$KVStore.put(block, slicesInBlock)$;

⑫ end for

⑬ return $chosenBlocks_{q_i}$.

步骤 2. HDFS 数据块过滤的步骤如算法 4 所示,算法 4 发生在 $InputFormat.getSplits$ 函数中. 首先初始化过滤后选定的数据块集合 $chosenBlocks_{q_i}$ 为空(行①),并读取算法 1 输出的临时文件,得到查询相关数据片的位置信息(行②). 然后根据输入文件列表得到所有需要处理的数据块(行③),对于每个数据块,判断其是否包含查询相关的数据片,如果包含则放入 $chosenBlocks_{q_i}$,否则抛弃(行④～⑧). 随后,为 $chosenBlocks_{q_i}$ 中每个选中的数据块建立一个 Key/Value 对,Key 为该数据块的 ID,Value 为该数据块内所有需要读取的数据片的偏移量,所有的 Key/Value 对保存在 HBase 一张临时表中(行⑨～⑫). 最后,返回查询相关的数据块 $chosenBlocks_{q_i}$(行⑬).

步骤 3. 数据块内部的数据片过滤发生在 $RecordReader.next$ 函数中,该函数读取步骤 2 中 HBase 中的临时表,得到在该块中所有需要读取的数据片的偏移量,然后在读取数据时跳过不需要读取的数据片. 如果某个数据片跨块存储,此时,假设其大部分存储在块 A 中,则让处理块 A 的 Mapper 处理该数据片,以最小化远程数据读取. 当查询为聚集值查询,则待 Hive 得到边界 GFU 结果后,需要与算法 3 中内部 GFU 的子结果合并得到最终结果.

如图 3 所示,假设有一个 SQL 查询,形式如图 3 所示. 1)先根据查询谓词,定位 DGFIndex 中的边界 GFU 与内部 GFU,因为该查询为聚集值查询,对于内部区域直接查询 HBase 中 *header* 得到子结果,保存于临时文件中. 对于边界区域,得到相关数据片在文件中的位置信息. 2)根据数据片在文件中的位置信息过滤与查询无关的数据块,然后为每个候选数据块保存需要读取的数据片的偏移. 3)在 Mapper 处理每个数据块时,跳过查询无关的数据片. 最终得到的子结果与第 1 步中的子结果合并,得到最终结果.

SELECT $sum(z)$
FROM *table*
WHERE $x>5$ AND $x<12$ AND $y\geqslant 12$ AND $y<16$

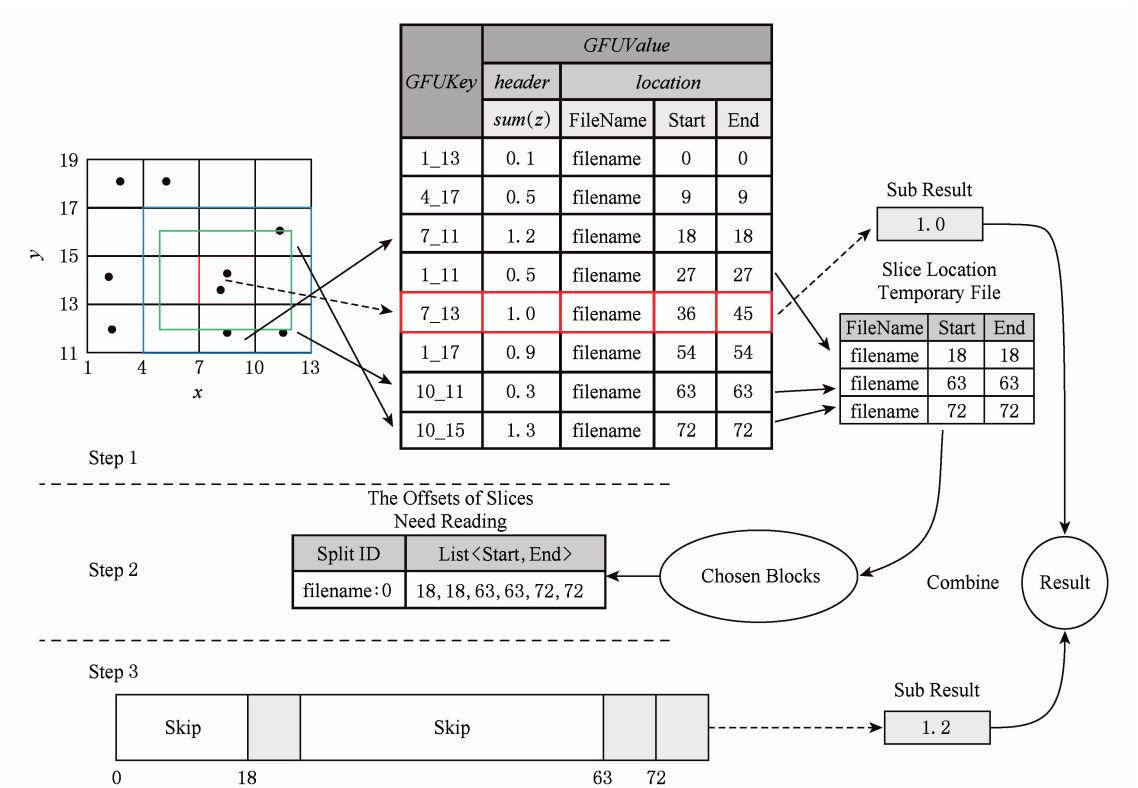


Fig. 3　An example of DGFIndex-based query.

图 3　DGFIndex 索引查询例子

2.4　DGFIndex 分割策略选择问题

由 2.2～2.3 节可以看出,分割策略的选择与查询性能息息相关:如果选择细粒度的分割策略,查询边界区域会较小,即读取的冗余数据量较小,而且解析数据、过滤数据的 CPU 开销也较小. 但是文献[6]指出,数据读取性能与数据片的大小成正比,即

此情况下数据读取性能较低;相反,如果选择粗粒度的分割策略,虽然此时的数据读取性能较高,但是由于查询边界区域会较大,造成读取过多冗余数据,而且,这也会造成解析数据、过滤数据的 CPU 开销较大. 图 4 展示了不同数据片大小对查询性能的影响(该实验的数据集使用存储格式为 RCFile 的 TPC-H lineitem 表,大小为 187 GB;查询使用 TPC-H 中的 Q6,实验环境与第 4 节中的描述一致,HDFS 数据块大小设置为 256 MB). 可以看出,对该查询而言,32 MB 为较优的数据片大小,过小或过大的数据片都无法得到最优的查询性能. 但是,由于不同的查询具有不同的查询谓词,即不同查询的最优数据片大小可能不同,这就造成查询间最优分割策略相互影响与制约. 因此,如何为 DGFIndex 选择最优的索引分割策略以最大化地提升查询集合性能成为一项重要且具有挑战性的任务.

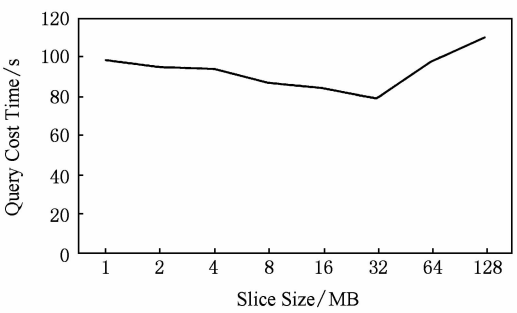


Fig. 4 Influence of different slice size on query performance.

图 4 不同数据片大小对查询性能造成的影响

3 基于代价估计的分割策略选择算法

3.1 问题描述

为了方便描述问题定义与代价模型,首先预定义一些形式化符号,如表 1 所示. $avgFieldSize_k$ 和 $avgRecordSize$ 的值可以通过运行涉及某个列的查询,然后根据 MapReduce Counter 的值得到.

基于多维查询集合的 DGFIndex 分割策略选择问题可以形式化定义为式(1)~(6):给定一个 Hive 中的表 T,其大小为 $tableSize$,给定一个在该表上的多维查询集合,由 n 个查询组成 $Q=\{q_1,q_2,\cdots,q_n\}$,该集合中每个查询的查询谓词包含至多 $idxNum$ 个查询维度,查询维度 k 的最小分割单位为 $minUnit_k$,例如整型维度的最小分割单位为 1,最小值和最大值分别为 $idx_k^{\min}$ 和 $idx_k^{\max}$. 现指定某种分割策略 SP_j,包含每个索引维度的分割区间大小 $SP_j=\{idxDI_1^j,idxDI_2^j,\cdots,idxDI_{idxNum}^j\}$,分割策略决定了数据片的大小,如式(5)所示,数据片的大小要小于 HDFS 数据块,否则会造成大量的网络远程读,降低数据读取性能. 通过建立基于索引进行数据处理的 MapReduce 代价模型 $CostModel(SP_j,q_i)$,可以得到每个查询在不同分割策略下的代价估计. 该问题的求解目标是为 DGFIndex 选择较优的分割策略,以使查询集合 Q 的总耗时最小.

$$\min \sum_{i=1}^{n} Time(q_i), \tag{1}$$

$$q\in Q:\{q_1,q_2,\cdots,q_n\},$$

$$\text{s.t. } Time(q_i)=CostModel(SP_j,q_i), \tag{2}$$

$$SP_j=\{idxDI_1^j,idxDI_2^j,\cdots,idxDI_{idxNum}^j\}, \tag{3}$$

$$\min Unit_k\leqslant idxDI_k^j, \tag{4}$$

$$blockSize>\frac{tableSize}{\prod_{k=1}^{idxNum}\frac{idx_k^{\max}-idx_k^{\min}}{idxDI_k^j}}, \tag{5}$$

$$1\leqslant k\leqslant idxNum. \tag{6}$$

Table 1 Definition of Symbols

表 1 符号定义

Symbol	Description
$idx_k^{\min}$	Minimum Value of Index Dimension k
$idx_k^{\max}$	Maximum Value of Index Dimension k
SP_j	Splitting Policy j
$idxDI_k^j$	Interval Size of Index Dimension k on Splitting Policy SP_j
$avgFieldSize_k$	Average Size of the kth Field in Record
$avgRecordSize$	Average Record Size
$qGFUKeySet_{q_i}^{SP_j}$	The q_i-related $GFUKey$ set on Splitting Policy SP_j
$qRelateData_{q_i}$	The q_i-related Data Size

3.2 代价模型

由问题描述可知,解决 DGFIndex 分割策略选择问题的关键为建立合理的代价模型以反映不同分割策略对查询的影响. MapReduce 代价模型已经被众多学者广泛研究[4,13-17],但是现有工作中的代价模型无法直接应用于本工作,原因如下:

1) 以往的代价模型在建模 Map 阶段数据读取代价时,直接使用 HDFS 块大小与固定数据读取吞吐的商,没有考虑不同数据片大小对读取性能的影响与索引对数据的过滤作用.

2) 当查询使用的 Mapper 数大于集群容量时,以往的代价模型直接使用 Map 阶段运行的遍数与单遍代价的乘积作为 Map 阶段的代价估计. 而在基于索引的查询处理时,由于每个 Mapper 读取数据量各不相同,数据量较小的 Mapper 可以快速执行

完,第 2 遍的 Mapper 可以马上使用空闲的 Mapper 资源进行执行,以往的代价模型无法准确描述该特性.

因此需要建立新的基于索引进行数据处理的 MapReduce 代价模型. 图 5 展示了 Hive 中操作符在 Map 阶段的数据处理流程及各操作符的作用. 不同分割策略只会影响 RecordReader 与 FilterOperator 的性能:对于 RecordReader,分割粒度越大其读取的数据量越大;对于 FilterOperator,分割粒度越大其数据解析与数据过滤的 CPU 开销越大. 而 FilterOperator 后输出的为符合查询谓词条件的记录,不同的分割策略并不会影响该值. 所以,只需要构建 Map 阶段数据读取与数据处理的代价模型,即可反映不同分割策略对查询集合性能的影响.

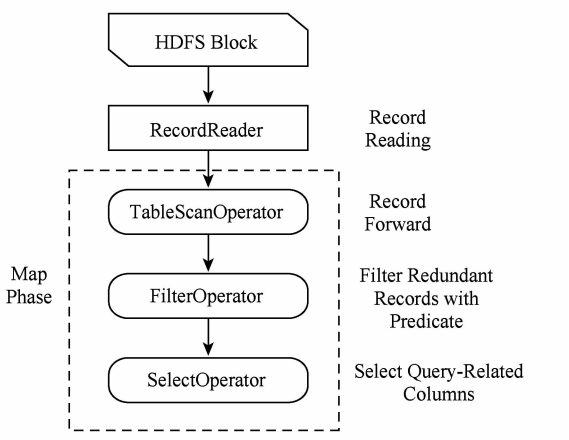


Fig. 5　Data process in Map phase.

图 5　Map 阶段数据处理流程

查询 q_i 在分割策略 SP_j 下,使用 DGFIndex 处理的代价估计建模为式(7),$MapRead(SP_j,q_i)$ 表示 Map 阶段数据读取的代价,$MapCPU(SP_j,q_i)$ 表示 Map 阶段数据解析与数据过滤的 CPU 代价.

$$CostModel(SP_j,q_i)=MapRead(SP_j,q_i)+MapCPU(SP_j,q_i). \quad (7)$$

Map 阶段数据读取的代价估计如式(8)所示,为查询相关的数据总量与数据读取吞吐的商,查询相关数据总量定义为式(9),数据片大小定义为式(10). 对每个查询而言,不同的分割策略 SP_j 导致了不同的查询相关的 $GFUKey$ 集合与不同的数据片大小,从而导致不同的查询相关数据量. 由于基于索引进行查询处理时,每个 Mapper 处理的数据量不同,因此处理数据量少的 Mapper 可以快速完成,这样未启动的 Mapper 可以继续使用空闲出来的资源进行处理. 并且,由于每次 Mapper 处理的次序与调度算法和集群中各节点的负载相关,因此很难事先得到 Mapper 处理的次序. 所以,这里使用读取查询相关全部数据的总耗时进行估计,这一点与以往的 MapReduce 代价模型不同. 对于列式存储,如 RCFile,只需要读取查询相关的列,所以读取的数据总量与查询相关列的总大小成正比.

$$MapRead(SP_j,q_i)=\frac{qRelatedData_{q_i}}{readThroughput(sliceSize_{SP_j},qRelatedData_{q_i})}. \quad (8)$$

$$qRelatedData_{q_i}=|qGFUKeySet_{q_i}^{SP_j}|\times sliceSize_{SP_j}\times\frac{\sum_{k=1}^{idxNum}avgFieldSize_k}{avgRecordSize}. \quad (9)$$

$$sliceSize_{SP_j}=\frac{tableSize}{\prod_{k=1}^{idxNum}\frac{idx_k^{\max}-idx_k^{\min}}{idxDI_k^j}}. \quad (10)$$

式(8)中数据读取吞吐的建模过程如下:由图 6,7 可以看出(该实验使用与图 4 中实验相同的数据集、集群环境与参数设置),数据读取吞吐与数据片大小成正比,与查询相关数据量成反比. 原因为当数据片增大时,随机读操作减少,数据的顺序读取性能提升;当查询相关数据量增大时,单节点启动的多个 Mapper 对磁盘 I/O 产生竞争,造成读取吞吐降低,这种情况尤其会出现在磁盘读取为瓶颈时,例如同一台物理机上的多虚拟机实例.

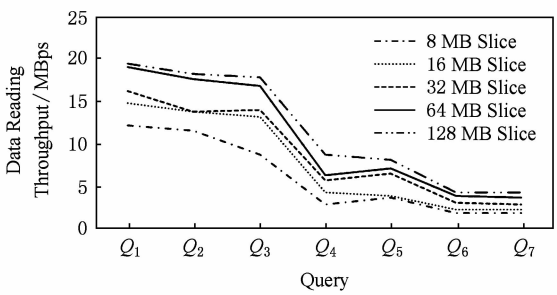


Fig. 6　Influence of query selectivity and slice size on data reading performance.

图 6　数据读取吞吐与查询选择度和数据片大小的关系

本文使用 Profiling 技术与多项式拟合方法建立数据读取吞吐与数据片大小和查询相关数据量之间的函数关系. 具体地,首先随机生成不同查询选择度的查询集合,然后通过指定不同的分割策略为数据表建立不同数据片大小的 DGFIndex,通过运行查询集合得到读取数据量大小与读取耗时(通过在 RecordReader 中添加 Counter 得到),从而得到对应的数据读取吞吐量,最后进行多项式函数拟合.

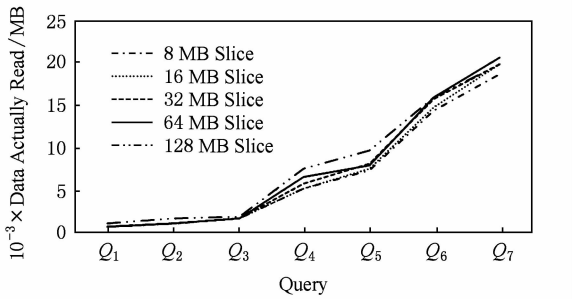


Fig. 7 Query related data size.

图 7 各查询相关数据量

此外,本文在建模数据读取吞吐时,如果有数据片为跨块读取,并没有考虑网络远程读取,原因有 2 点:1)因为使用了存储大部分数据片的 Mapper 处理该数据片的优化技术,远程读取数据量较小,通过实验发现在使用 8～128 MB 数据片大小时,远程读取数据量约占总读取数据量的 0～6%,因此对代价估计的影响有限. 2)因为对于列式存储格式而言,如 RCFile,数据片为其中的一个 Row Group,而 Row Group 为最小读取单元,所以较难统计读取部分 Row Group 的时间.

Map 阶段的数据解析与数据过滤的 CPU 代价估计如式(11)所示,即读取的记录数与每条记录消耗的 CPU 代价的乘积. 经过实验发现,每条记录消耗的 CPU 代价为常量,不受其他变量的影响,在本文使用的实验环境中,该常量值约为 3167 ns/record.

$$MapCPU(SP_j,q_i)=\frac{|qGFUKeySet_{q_i}|\times sliceSize}{avgRecordSize}\times CPU\,Cost. \quad (11)$$

由 3.2 节的代价模型可以看出,首先,分割策略在 2 方面决定了 Hive 处理查询时读取的数据量:1)查询相关的数据片的数量,即给定索引维度的分割区间大小与查询谓词,就可以得到该查询相关的数据片数量;2)数据片大小,一旦各索引维度分割区间确定,就确定了数据片大小. 其次,分割策略决定了数据读取吞吐,给定数据片大小与查询相关数据量,就可以根据拟合的函数得到数据读取吞吐. 最后,分割策略还决定了 Map 阶段数据解析与过滤的 CPU 开销,因为其确定了读取数据量,也就确定了记录数.

由此可知,一旦给定分割策略 SP_j,就可以对查询集合 Q 中的每个 q_i 进行代价估计,从而衡量分割策略的优劣. 但是,由于候选分割策略的搜索空间大小正比于多个维度可选方案的乘积,并且对于每种分割策略,都要计算整个查询集合的代价估计,因此遍历搜索最优的分割策略将会花费较长的时间. 为了加快搜索过程,本文提出了一种基于模拟退火的两阶段搜索算法来寻找较优的分割策略.

3.3 基于模拟退火的两阶段分割策略搜索算法

由于分割策略选择过程为多个索引维度分割区间选择组合问题,为了避免搜索陷入局部最优解,本文选择模拟退火算法进行搜索. 基于模拟退火的分割策略搜索算法分为 2 个阶段:粗粒度近似搜索与细粒度精确搜索. 第 1 阶段找到近似最优解,第 2 阶段减小搜索步长,在近似最优解周围搜索精确"最优解". 这里的最优解并非实际的最优分割策略,因为这取决于代价模型的准确性,而代价模型本身为估算值.

算法过程如算法 5 所示:

算法 5. 基于模拟退火的两阶段搜索算法.

输入:查询集合 Q;

输出:较优的分割策略.

① $minIntrls=currentItrls=getInitialInterval()$;
② $minCost=currentCost=getCost(currentIntrls)$;
③ while $temp>1$ do
④ $newIntrls=getNeighbor(currentIntrls,temp)$;
⑤ $newCost=getCost(newIntrls)$;
⑥ if $newCost<currentCost$ then
⑦ $acceptRatio=1$;
⑧ else
⑨ $acceptRatio=e^{\frac{newCost-currentCost}{temp}}$;
⑩ end if
⑪ if $acceptRatio>random()$ then
⑫ $currentIntrls=newIntrls$;
⑬ $currentCost=newCost$;
⑭ if $currentCost<minCost$ then
⑮ $minIntrls=currentIntrls$;
⑯ end if
⑰ end if
⑱ $temp^*=(1-coolingRate)$;
⑲ end while
⑳ return $minIntrls$.

首先初始化当前的分割策略与代价估计,并初始化最优分割策略与最优代价估计为当前分割策略与对应的代价(行①～②). 随后,开始迭代,当温度 $temp>1$ 时,重复迭代过程(行③). 在每次迭代时,首先得到当前分割策略的邻居分割策略(详细过程

会在算法 6 中介绍)(行④),然后使用代价模型计算邻居分割策略的查询集合代价估计(行⑤). 行⑥～⑩计算接受邻居分割策略的概率:如果邻居分割策略得到的代价估计比当前的分割策略的代价估计小,则接受概率为 1;否则,计算一个概率值,表示是否接受比当前最优分割策略坏的方案,该概率值与温度成正比,即温度高时,接受较差的分割方案的概率较大,当温度较低时(即将要结束迭代时),不易接受较差的分割策略. 如果接受概率大于某个随机数,则接受邻居分割策略,设置当前的分割策略与代价估计为邻居分割策略及对应代价估计(行⑪～⑬). 如果当前的分割策略的代价估计小于目前得到的最优分割策略,则设置最优分割策略为当前的分割策略(行⑭～⑯). 最后使用冷却速率 *coolingRate* 更新温度值,重复迭代过程(行⑱～⑲). 待迭代结束后,返回得到的分割策略(行⑳).

在模拟退火迭代过程中,需要得到当前分割策略的邻居,该过程逻辑如算法 6 所示. 首先拷贝当前策略(行①),然后根据当前的温度 *temp* 确定搜索步长:如果 $temp>10$,则进行粗粒度(行②～③);如果 $temp<10$,则进行细粒度搜索,即此时对各索引维度改变的步长为粗粒度时的 10%(行④～⑥). 对于每个索引维度,随机地增加或减少其分割区间大小,增加或减少的步长为随机数(行⑦～⑭).

算法 6. 得到当前分割策略的邻居算法.

输入:当前的分割策略 *currentIntrls*、当前的温度 *temp*;

输出:当前分割策略的邻居.

① $newIntrls=clone(currentIntrls)$

② if $temp>10$ then

③　$ratio=1$;

④ else

⑤　$ratio=0.1$;

⑥ end if

⑦ for kth *intrl* in $newIntrls(k$ from 1 to $idxNum)$ do

⑧　$random=random()$;

⑨　if $random>0.5$ then

⑩　$intrl_k+=\frac{idx_k^{\max}-idx_k^{\min}}{minUnit_k}\times random()\times ratio$;

⑪　else

⑫　$intrl_k-=\frac{idx_k^{\max}-idx_k^{\min}}{minUnit_k}\times random()\times ratio$;

⑬　end if

⑭ end for

⑮ return *newIntrls*.

从算法 6 过程可以看出,迭代次数由初始温度 *temp* 与冷却速率 *coolingRate* 决定:*temp* 越小,*coolingRate* 越大,迭代次数越少,收敛速度越快,但是可能得不到最优解;反之,则可能增加算法搜索时间. 在下面的实验中,设置 $temp=200$,$coolingRate=0.01$.

4　实验结果与分析

4.1　实验环境与测试集

1) 硬件环境. 本次实验使用由 32 个虚拟机节点组成的集群环境,每个虚拟机节点具有 12 核 CPU、26 GB 内存、600 GB 磁盘. 其中一个节点作为 MapReduce 计算框架的主节点 JobTracker,一个节点作为 HDFS 的主节点 NameNode,一个节点作为 HBase 的主节点 HMaster. 其他节点作为从节点,运行 DataNode,TaskTracker. Hive 与 NameNode 运行在同一节点.

2) 软件环境. 每个节点使用 CentOS 7.0,JDK 1.7.0_65 64bit. 使用 Hadoop-1.2.1,HBase-0.94.23(用于存储 DGFIndex 的索引表). DGFIndex 实现在 Hive-0.14.0 中. HDFS 块大小设置为 256 MB,Hadoop,Hive 与 HBase 的其他配置参数都使用默认值.

3) 测试集. 本实验使用 TPC-H 测试集中的 lineitem 表与 Q6,使用被广泛使用的 RCFile 与 ORC 作为文件存储格式,不同数据集大小如表 2 所示. 本实验为 lineitem 表在 l_discount,l_quantity 和 l_shipdate 上建立三维索引,各维度的基数如表 3 所示. 查询集合由随机生成的 Q6 构成,集合中的查询具有不用的选择度,本实验使用 2 个查询集合:30 个随机查询构成的集合 QSet30 与 50 个随机查询构成的集合 QSet50. 选择该测试集的原因有 3 点:1)TPC-H 被广泛地用于数据仓库类系统的评测工作,因此使用该测试集的结果具有可比性;2)由于在 Hive 中,索引只作用于过滤单表读取时查询无关的数据,所以选择单表数据集足以评测索引的数据过滤性能,并且 lineitem 表是该测试集中最大的表;3)Q6 是 TPC-H 中典型的多维查询,可以充分测试 DGFIndex 的多维数据索引能力. 此外,在本实验中,在运行每个查询前都会清空操作系统的缓存并

重启 Hadoop,以保证索引定位的数据从磁盘读取,从而避免缓存对结果造成的影响.并且,查询集合中每个查询都会运行 3 次,在结果中使用 3 次结果的平均值,以减弱集群不稳定对结果造成的影响.

Table 2 Data Size

表 2 实验数据与大小 GB

Data Set	RCFile	ORC
Data-Small	94	58
Data-Medium	188	115
Data-Large	460	282

Table 3 Cardinality of Index Dimensions

表 3 索引维度基数

Index Dimension	Cardinality
l_discount	11
l_quantity	50
l_shipdate	2 526

4.2 DGFIndex 的性能

本实验中,在 RCFile 上创建 Hive 原生索引 Compact Index 与 DGFIndex,此外,本实验还与 ORC[6] 中的索引进行对比. ORC 是 Hive 目前最新的列式存储格式,内嵌了索引功能. Compact Index 与 ORC 中索引的数据过滤效果与索引维度数值在文件中的分布有关,对于均匀数据集 TPC-H 来说,两者的性能较差. 为了提升两者的索引性能,事先使用并行 Order By 对两者的数据表在索引维度上进行全排序预处理. 图 8,9 展示了在不同数据集大小与不同查询选择度下的各种索引的查询性能.

从实验结果可以看出,相对不使用任何索引的 RCFile,Compact Index 可以提升 2～7 倍查询性能,DGFIndex 可以提升 4.7～15 倍查询性能,并

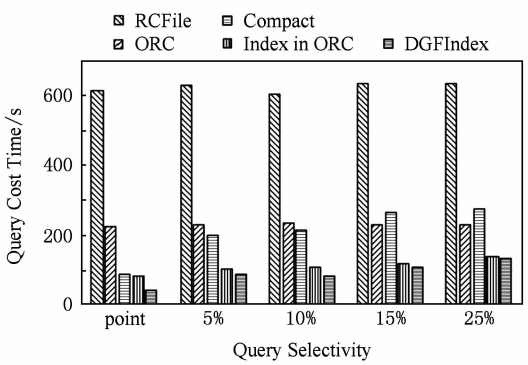


Fig. 8 Query cost time of Data-Medium.

图 8 Data-Medium 上的查询耗时

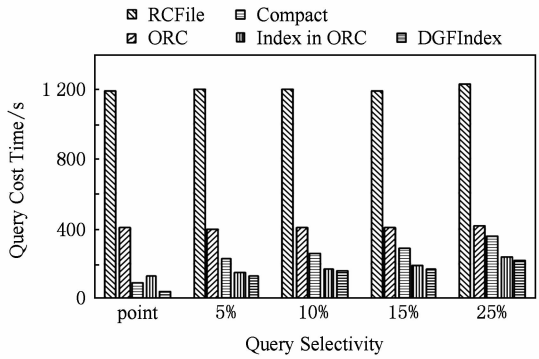


Fig. 9 Query cost time of Data-Large.

图 9 Data-Large 上的查询耗时

且,DGFIndex 比 Compact Index 的查询性能提升了 50%～114%. 原因有 2 点:1)Compact Index 只能在 HDFS 数据块级别过滤查询无关数据,这会造成读取过多的冗余数据;而 DGFIndex 可以在细粒度的数据片级别过滤查询无关的数据,从而大幅降低冗余数据的读取,提升查询性能. 2)Compact Index 读取索引的方式为启用额外的 MapReduce 任务全表扫描的方式,索引读取效率较低;而 DGFIndex 使用基于键值的方式,只需读取查询相关的键,读取效率较高. 此外,相比于不使用索引的 ORC,其内的索引提升 1.6～2.6 倍查询性能. DGFIndex 比 ORC 中的索引查询性能提升了 8%～28%,而对于点查询来说,分别提升了 1.2 倍和 2.2 倍. 可以看出,DGFIndex 在低选择度时的优势更明显,原因为:经过索引定位后,DGFIndex 使用了比 ORC 更少的 Mapper.

4.3 分割策略选择算法的有效性

本实验使用 RCFile 作为底层存储格式. 图 10,11 展示了不同数据量下基于代价估计的分割策略选择算法得到的分割策略的查询集合耗时与人工指定不同数据片大小的分割策略的查询集合耗时对比

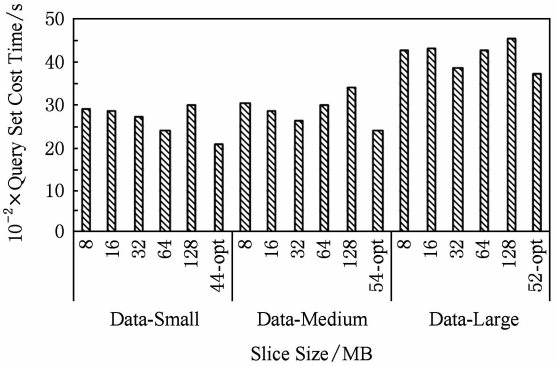


Fig. 10 Cost time of QSet30.

图 10 QSet30 在不同数据片大小下的查询耗时

结果,其中后缀为 opt 的为本文提出算法选择的分割策略对应的数据片大小. Data-Small 下人工指定的分割策略(对应索引维度 l_quantity,l_discount,l_shipdate)分别为[2,0.01,60],[4,0.01,60],[4,0.01,120],[4,0.02,115]与[8,0.02,115]. Data-Medium 下人工指定的分割策略分别为[2,0.01,29],[2,0.01,58],[4,0.01,59],[8,0.01,58]与[12,0.01,73]. Data-Large 下人工指定的分割策略为[1,0.01,24],[2,0.01,24],[2,0.01,47],[4,0.01,52]与[4,0.01,97]. 由本文算法得到的较优分割策略如表 4 所示.

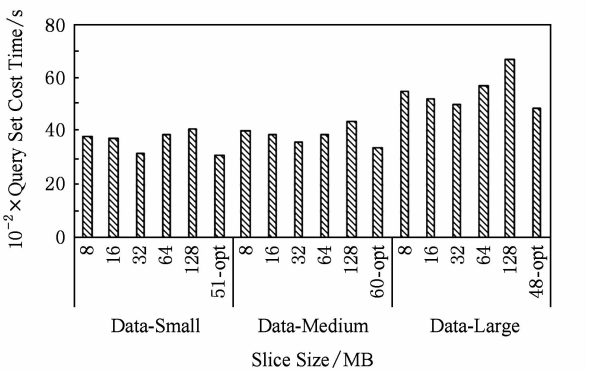


Fig. 11　Cost time of QSet50.

图 11　QSet50 在不同数据片大小下的查询耗时

Table 4　Optimal Splitting Policy

表 4　本文算法得到的较优分割策略

Data Set	Data-Small	Data-Medium	Data-Large
QSet30	[6,0.01,115]	[5,0.01,78]	[2,0,01,78]
QSet50	[9,0.01,83]	[9,0.01,49]	[3,0.01,49]

由结果可以看出,人工指定分割策略方法在选择不同分割策略时性能各有差异,并且索引粒度太大或太小都无法得到较优的性能,因此在用户不熟悉数据与查询特征时较难选择较优的分割策略. 相反,基于代价估计的分割策略选择算法可以根据查询集合自动地得到较优的分割策略,从结果可以看出与人工指定方法相比,最多可以减少查询集合耗时 30%.

4.4　基于代价估计的分割策略选择算法收敛速度

图 12 展示了基于遍历算法、模拟退火与人工选择分割策略选择算法的耗时,这里假设人工选择的耗时为 1 s.

从图 12 可以看出,基于模拟退火的算法比遍历算法快 12～24 倍,在短时间内搜索到较优的索引分割策略. 此外,基于遍历算法的分割策略选择算法的

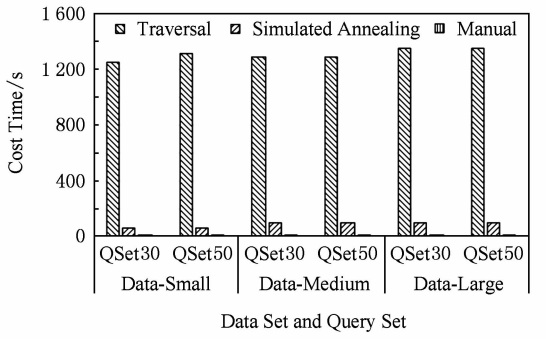


Fig. 12　Cost time of splitting policy search algorithm.

图 12　分割策略选择算法耗时

耗时与多个索引维度基数的乘积成正比,因为其需要遍历每一种可能的分割策略的可行性. 而在本实验中,索引维度的基数都较小,如果应用中遇到更大基数的索引维度,如浮点类型维度,则遍历算法的耗时是不可接受的. 如果根据数据与查询特征人为选取分割策略,虽然可以快速选定,但是较难选择较优的分割策略.

5　结束语

本文针对采集类数据与查询特征,提出了一种面向 Hive 的多维索引技术—DGFIndex,该索引以细粒度过滤查询无关的数据大幅减少冗余数据读取,其查询性能比 Hive 原生索引 Compact Index 提升 50%～114%,比 ORC 中的索引提升 8%～28%,并极大地提高了点查询的性能.

但是在创建 DGFIndex 时,需要人为指定各索引维度分割区间大小,在对查询与数据特征不熟悉情况下,用户很难选择最优的索引分割策略. 针对该问题,本文首先提出了一种新的 MapReduce 代价模型,基于该模型提出了一种基于模拟退火的两阶段分割策略搜索算法,实验证明该算法可以在较短的时间内得到较优的分割策略.

本文提出的代价模型没有考虑数据分布维度,在数据分布不均匀时,对 DGFIndex 而言,各数据片的大小会出现不同. 在这种情况下,使用查询相关 *GFUKey* 的数量估算查询相关数据量会不精确,需要得到数据片在文件中的布局来估算该值. 但是由于需要记录更多的信息,因此代价估计计算的速度会减慢,分割策略的搜索速度也会大大减慢. 在未来的工作中,我们将会在代价模型中引入数据分布维度,并优化在该种情况下分割策略搜索的速度.

参 考 文 献

[1] Zaharia M, Chowdhury M, Franklin M, et al. Spark: Cluster computing with working sets [C] //Proc of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 2010). Berkeley, CA: USENIX Association, 2010: 10

[2] Hu Songlin, Liu Wantao, Rabl T, et al. DualTable: A hybrid storage model for update optimization in Hive [C] // Proc of the 31st IEEE Int Conf on Data Engineering (ICDE 2015). Piscataway, NJ: IEEE, 2015: 1340-1351

[3] Liu Yue, Hu Songlin, Rabl T, et al. DGFIndex for smart grid: Enhancing Hive with a cost-effective multidimensional range index [C] //Proc of the 40th Int Conf on Very Large Data Bases (VLDB 2014). New York: ACM, 2014: 1496-1507

[4] Wang Yue, Xu Yingzhong, Liu Yue, et al. QMapper for smart grid: Migrating SQL-based application to Hive [C] // Proc of the ACM SIGMOD Int Conf on Management of Data (SIGMOD 2015). New York: ACM, 2015: 647-658

[5] Thusoo A, Sarma J S, Jain N, et al. Hive: A warehousing solution over a map-reduce framework [C] //Proc of the 35th Int Conf on Very Large Data Bases (VLDB 2009). New York: ACM, 2009: 1626-1629

[6] Huai Yin, Ma Siyuan, Lee Rubao, et al. Understanding insights into the basic structure and essential issues of table placement methods in clusters [C] //Proc of the 40th Int Conf on Very Large Data Bases (VLDB 2014). New York: ACM, 2014: 1750-1761

[7] Jiang Dawei, Ooi B C, Shi Lei, et al. The performance of MapReduce: An in-depth study [C] //Proc of the 36th Int Conf on Very Large Data Bases (VLDB 2010). New York: ACM, 2010: 472-483

[8] Dittrich J, Quiane-Ruiz J A, Jindal A, et al. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing) [C] //Proc of the 36th Int Conf on Very Large Data Bases (VLDB 2010). New York: ACM, 2010: 515-529

[9] Eltabakh M, Ozcan F, Sismanis Y, et al. Eagle-eyed elephant: Split-oriented indexing in Hadoop [C] //Proc of the 16th Int Conf on Extending Database Technology (EDBT/ICDT 2013). New York: ACM, 2013: 89-100

[10] Richter S, Quiane-Ruiz J A, Schuh S, et al. Towards zero-overhead static and adaptive indexing in Hadoop [J]. The VLDB Jounal, 2014, 23(3): 469-494

[11] Aji A, Wang Fusheng, Vo H, et al. Hadoop GIS: A high performance spatial data warehousing system over MapReduce [C] //Proc of the 39th Int Conf on Very Large Data Bases (VLDB 2013). New York: ACM, 2013: 1009-1020

[12] Eldawy A, Mokbel M. A demonstration of spatial Hadoop: An efficient MapReduce framework for spatial data [C] // Proc of the 39th Int Conf on Very Large Data Bases (VLDB 2013). New York: ACM, 2013: 1230-1233

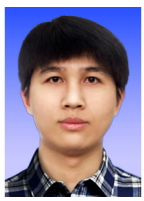
[13] Herodotou H. Hadoop performance models [R]. Pittsburgh, PA: arXiv preprint, 2011

[14] Herodotou H, Badu S. Profiling, what-if analysis, and cost-based optimization of MapReduce programs [C] //Proc of the 37th Int Conf on Very Large Data Bases (VLDB 2011). New York: ACM, 2011: 1111-1122

[15] Lin Xuelian, Meng Zide, Xu Chuan, et al. A practical performance model for Hadoop MapReduce [C] //Proc of 2012 IEEE Int Conf on Cluster Computing (Cluster 2012). Piscataway, NJ: IEEE, 2012: 231-239

[16] Wang Youwei, Wang Weiping, Meng Dan. Query optimization by statistical approach for Hive data warehouse [J]. Journal of Computer Research and Development, 2015, 52(6): 1452-1462 (in Chinese)
(王有为, 王伟平, 孟丹. 基于统计方法的 Hive 数据仓库查询优化实现[J]. 计算机研究与发展, 2015, 52(6): 1452-1462)

[17] Song Jie, Li Tiantian, Zhu Zhiliang, et al. Research on I/O cost of MapReduce join [J]. Journal of Software, 2015, 26(6): 1438-1456 (in Chinese)
(宋杰, 李甜甜, 朱志良, 等. MapReduce 连接查询的 I/O 代价研究[J]. 软件学报, 2015, 26(6): 1438-1456)



Liu Yue, born in 1988. PhD candidate. Student member of China Computer Federation. His research interests include distributed system and database system.

Li Jintao, born in 1962. Professor and PhD supervisor. His research interests include multimedia technology, virtual reality technology, and pervasive computing technology.

Hu Songlin, born in 1973. Professor and PhD supervisor. Senior member of China Computer Federation. His research interests include service computing, distributed system and middleware.