软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(3):502−513 [doi: 10.13328/j.cnki.jos.005161]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种面向 HDFS 的多层索引技术*

何 龙[1], 陈晋川[1,2], 杜小勇[1,2]

[1](中国人民大学 信息学院,北京 100872)

[2](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)

通讯作者: 陈晋川, E-mail: jcchen@ruc.edu.cn

摘 要: SOH(SQL over HDFS)系统通常将数据存储于分布式文件系统 HDFS(Hadoop distributed file system)中,采用 Map/Reduce 或分布式查询引擎来处理查询任务.得益于 HDFS 以及 Map/Reduce 的容错能力和可扩展性,SOH 系统可以很好地应对数据规模的飞速增长,完成分析型查询处理.然而,在处理选择型查询或交互式查询时,这类系统暴露出了性能上的缺陷.提出一种通用的索引技术,可以应用于 SOH 系统中,以提高其查询处理的效率.分析了 SOH 系统访问 HDFS 文件的过程,指出了其中影响数据加载时间的关键因素.提出了 split 层和 split 内部双层索引机制;设计并实现了聚集索引和非聚集索引;最后,在标准数据集上进行了大量实验,并与现有基于 HDFS 的索引技术进行了比较.实验结果表明,所提出的索引技术可以有效地提高查询处理的效率.

关键词: 查询;索引;HDFS;SOH;大数据

中图法分类号: TP311

中文引用格式: 何龙,陈晋川,杜小勇.一种面向 HDFS 的多层索引技术.软件学报,2017,28(3):502−513. http://www.jos.org.cn/1000-9825/5161.htm

英文引用格式: He L, Chen JC, Du XY. Multi-Layered index for HDFS-based systems. Ruan Jian Xue Bao/Journal of Software, 2017,28(3):502−513 (in Chinese). http://www.jos.org.cn/1000-9825/5161.htm

Multi-Layered Index for HDFS-Based Systems

HE Long[1], CHEN Jin-Chuan[1,2], DU Xiao-Yong[1,2]

[1](School of Information, Renmin University of China, Beijing 100872, China)

[2](Key Laboratory of Data Engineering and Knowledge Engineering, MOE (Renmin University of China), Beijing 100872, China)

Abstract: The SOH (SQL over HDFS) systems usually store the data into distributed file system HDFS (Hadoop distributed file system), and process queries by the Map/Reduce computing framework or distributed database query engine. Benefitting from the fault tolerance and scalability provided by Map/Reduce and HDFS, SOH systems perform well in processing analytical queries over big data. However, the efficiency of such systems is too low to meet the requirement of selective queries or interactive queries which have strict limit on the query response time. This paper proposes a HDFS-based index, called HIndex, for SOH systems. HIndex can easily be integrated into the existing SOH systems to improve the efficiency of query evaluation. The process that SOH systems access data stored in HDFS is analyzed, and the important factors affecting the time cost is highlighted, a two-layer index structure is proposed, and both aggregated and non-aggregated index techniques are implemented. According to the experiments conducted on standard datasets, HIndex performs much better than Hadoop++, a state-of-the-art HDFS-based index.

Key words: query; index; HDFS; SOH; big data

* 基金项目: 国家重点研发计划(2016YFB1000702); 中国人民大学预研委托(团队)基金(14XNLQ06); 国家自然科学基金(61003086)

Foundation item: National Key Research and Development Plan (2016YFB1000702); Research Funds of Renmin University of China (14XNLQ06); National Natural Science Foundation of China (61003086)

收稿时间: 2016-07-29; 修改时间: 2016-09-14; 采用时间: 2016-11-01; jos 在线出版时间: 2016-11-29

CNKI 网络优先出版: 2016-11-29 13:35:10, http://www.cnki.net/kcms/detail/11.2560.TP.20161129.1335.011.html

在过去的 10 多年间,随着大数据的兴起,涌现出众多的 SOH(SQL over HDFS)系统.早期的 SOH 系统是在 Hadoop 系统基础上提供 SQL 语言的支持,如 Hive,HBase 等,其技术特点可以简单概括为 Map/Reduce+HDFS.即:采用 HDFS 存储数据,并依赖于 Map/Reduce 计算框架处理查询.近几年来出现了以 Impala,Presto 为代表的新一代 SOH 系统.它们的特点是采用分布式查询引擎代替了 Map/Reduce 框架,因此极大地提升了查询处理速度.

SOH 系统通常被用于存储离线数据,处理分析型查询,比如存储交易系统的日志,并计算一段时间内符合某个条件的交易数量、金额等.分析型查询对响应时间的要求较为宽松,主要挑战在于数据的规模.

由于 HDFS 以及 Map/Reduce 优良的可扩展性和容错能力,SOH 系统可以很好地通过水平扩展来应对数据规模的飞速增长.在实际应用中,很多大型企业,如谷歌、Facebook、百度、阿里等,都采用 SOH 系统来管理数据,其集群的规模往往达到几千个节点.SOH 系统已经在分析型查询领域取得了极大的成功.

在实际应用中,除了分析型查询,我们也经常需要处理交互式查询或选择型查询(selective queries).这类查询的特点是对查询响应时间有较为严格的要求,比如在一个电子商务系统中,用户希望查看自己在过去一个星期内的交易记录,此时,系统必须在一个合理的时间范围内响应查询,否则将造成不良的用户体验.

传统的数据库管理技术中,提高查询处理速度最常用的方法是索引.通过索引,快速过滤不符合查询要求的数据,可以极大地降低 I/O,缩小搜索范围,降低响应时间.然而,传统的索引技术并不能直接应用到 SOH 系统中.传统索引需要记录数据项在磁盘上的位置,这样在处理查询时就可以精确定位所需要的数据,避免额外的 I/O.然而,HDFS 提供的是一种透明的访问机制,应用程序不能事先定位某个数据项在磁盘上的位置.为了实现容错,HDFS 需要为一个文件存储多个备份.当程序访问 HDFS 上某个文件时,HDFS 会自动选择该文件的某个备份,将文件块(block)组织为若干个 split,并以 split 为单位将文件传递给发出请求的程序.

ORC(optimized row columnar) File[1]最早尝试在 HDFS 文件中增加简单的索引机制,该技术发源于早期 Hive 系统中的 RC File[2],综合了行存储和列存储的优点.ORC File 实现了轻量级的索引,主要包括:(1) 文件层面的统计;(2) Stripe 层面的统计;(3) 索引组层面的统计.Parquet(Parquet.https://parquet.apache.org/)的思想来源于 Google 的 Dremel[3],是面向分析型查询的列式存储结构.Parquet 采用了记录碎片化与组装算法,支持高效的压缩算法和序列化.Parquet 可以通过文件内部的一些统计信息,跳过不符合条件的数据.Hadoop++[4]是由德国 Saarland 大学 Dittrich 等人提出的基于 HDFS 的索引技术,Hadoop++采用注入式的修改,并不改变 Hadoop 原本的架构,而是使用 UDF(用户定义函数)来达到在数据载入阶段使用索引进行快速定位的目的.其主要实现方式是在 Hadoop 本身划分好的每个 split 数据之后加入索引.每次在读取扫描的时候,先读取索引并定位到满足条件的记录,只读取满足条件的记录,无需读取整个 split.通过上述方式,实现减少磁盘 I/O、提高查询性能的目的.HAIL[6]是该研究组的后续成果,在 Hadoop++的基础上提供两种索引机制:static index 和 adaptive index.与 Hadoop 相比,Hadoop++具备明显的性能优势.然而,Hadoop++没有考虑在 split 层进行过滤,也不能支持非聚集索引.本文工作在 Hadoop++的基础上解决了上述两个问题.我们提出了两层的索引机制:在 split 层建立了属性值区域的统计信息,实际是简化的直方图,根据这一统计信息,我们可以判断一个 split 是否包含符合查询要求的数据;在 split 内部,我们设计并实现了聚集和非聚集索引,可以更全面地支持不同的查询.

上述技术(包括本文工作)都将索引存储在 HDFS 文件内部.另一类基于 HDFS 的索引技术尝试将索引存储于外部系统,比如关系数据库中,我们将其称为基于混合系统的索引技术.Eagle-Eyed Elephant(E3)[6]系统提出一种主存适应缓存方法来维护一个倒排索引表.该倒排索引表存放于 DBMS 中,Hadoop 执行查询时,在 DBMS 中,先通过倒排索引表确定出哪些 split 是查询需要的,然后再在 HDFS 中扫描相关 split.威斯康星大学的 Gankidi 等人在微软的 Polybase 的系统(PDW)上实现了一个索引[7],其主要方法是:在并行 DBMS 和 MapReduce 混合系统中,对于存储在 HDFS 的数据表的属性建索引.索引以表的方式存放在并行 DBMS 中,查询时先查询 DBMS 中的索引表,得到相应的 block,然后再通过 Map/Reduce 对其相应的 block 进行处理.ScalaGiST[8]是由浙江大学的 Lu 等人提出,其主要思想是独立于 Hadoop 之外实现了一套分布式的索引服务系统.ScalaGiST 是应用在动态集群环境下的可扩展的通用的搜索树.

上述基于混合系统的索引的实现方式需要依赖于 HDFS 之外系统的支持,加大了开发和维护的负担.此外,

SOH 系统在使用这类索引时需要更改其现有的查询处理逻辑(先检索外部系统的索引).本文所提出的索引可以无缝地集成到现有 SOH 系统中,它将根据查询条件自动过滤数据,不会修改查询处理过程.

近年来,一些公司相继推出实时交互式查询处理系统,代表系统包括 Spark,Dremel 以及 Puma.Spark SQL[9]是基于 Spark 计算引擎的查询系统.Spark 是基于内存的计算引擎,通过一个抽象数据集,在内存中对数据进行操作.Spark 效率远远高于 Map/Reduce,其缺陷在于数据存储于 HDFS,数据载入效率较低.Dremel[3]是 Google 的一个可扩展的交互式数据分析系统,通过将多层执行树和列存储数据绑定,实现百万条记录秒级别的聚合查询处理.Dremel 采用了更灵活的嵌套数据模型,并与列存储相结合,实现了对数据的高效扫描.Puma 是 Facebook 公司的一个实时数据分析系统.Puma 系统[10]使用标准的 MapReduce 框架从 Hive 中读取数据,并在批处理环境中运行流处理应用程序.

本文的侧重点是提出一种适用于 SOH 系统的索引(HIndex),并非一类新型的查询处理系统,因此本文工作和上述实时查询处理系统是正交的.此外,HIndex 还可以集成于部分实时查询处理系统中,提高其数据访问效率.比如:可以在 Spark SQL 的数据接入层中集成 HIndex,提高查询性能.

本文首先详细分析了 SOH 系统访问 HDFS 文件的过程,指出了其中影响数据加载时间的关键因素.在此基础上,我们提出了两层索引机制 HIndex,即:split 层索引和 split 内部索引.我们还设计并实现了聚集索引和非聚集索引以及相应的查询处理算法.最后,我们通过大量实验验证了所提出索引的有效性.相对于 Hadoop++,我们的索引技术可以将查询响应时间缩短约 40%左右;相对于 Hadoop,性能提升约 7 倍.

1 HDFS 文件读取过程分析

本节将分析现有 SOH 系统从 HDFS 加载数据的过程,找出影响数据加载时间的主要因素,以有针对性地进行优化.

1.1 数据存储

在执行一个计算任务之前,首先要将数据存储到 HDFS 中.HDFS 将数据文件划分为多个数据块(block),每个 block 有多个备份,每个备份依照一定的策略存放于不同的数据节点上.HDFS 默认是两个副本,即,一个 block 在 HDFS 上共有 3 份且分别存放在不同的数据节点上.如图 1 所示:数据文件 *F* 通过 *block* 函数被划分为 3 个 block,通过 Fetch 和 Store 操作符,分别将不同的 block 存储于不同的数据节点.为了便于描述,本文假定每个 block 只有一个副本,即:每个 block 有两份存储于 HDFS 中,分别存放于不同的数据节点中.

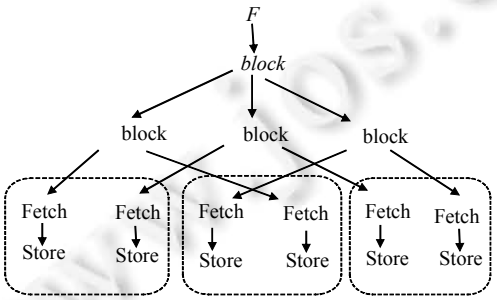


Fig.1 Structure of a HDFS file

图 1 文件在 HDFS 中的存储方式

1.2 数据读取

数据读取主要包含两个阶段:逻辑划分阶段和 Map Task 处理阶段.

- 在逻辑划分阶段,对于 HDFS 上需要处理的数据子集,即一个或者多个 block,将它们划分为 split.split 是逻辑划分,它可以只包含一个或多个 block,这个可以通过用户定义函数来定义,通过指定划分 split

的大小来确定该 split 包含几个 block.系统通过 split 包含的 block 元信息,定位到 HDFS 文件的相应偏移量处;

- 在 Map Task 处理阶段,系统会启动 Map Task,每个 Map Task 通过用户自定义函数 *itemize* 将 split 划分为一个个数据项,然后由 RecordReader 将数据项读取出来.最后系统对于每个数据项通过 map task 中的相应运算函数以流水线的方式进行处理.

如图 2 所示,左半部分表示的是一个 split 包含两个 block 的处理流程,右半部分是只包含一个 block 的处理流程.

下面我们分析决定数据加载成本的主要因素.逻辑划分阶段发生于 SOH 系统的主节点中,其主要是内存计算,并且计算量并不大.因此,相对于其他方面,逻辑划分阶段对数据加载时间的影响可忽略不计.

Map Task 处理阶段代价主要产生于 3 部分:RecordReader 读取数据产生的磁盘 I/O、Map Task 初始化以及操作符的运算.RecordReader 读取数据产生的磁盘 I/O 是一个计算任务最主要的开销.大量的数据扫描导致产生了大量的磁盘 I/O,加剧了计算任务的执行时间.前文已经提到:包括 Hadoop++,ORC File 等,均是通过过滤不必要的数据以减少磁盘 I/O.

Map Task 初始化的代价主要是启动 JVM 虚拟机产生的代价,此外还包括运行环境初始化的时间开销.由于 SOH 系统面向于大数据,处理的数据量非常大,Map Task 数目一般都很多,因此,此部分的代价是不容忽视的.尤其是 Hadoop++,ORC File 等系统中已经对磁盘 I/O 进行了优化,此时,Map Task 初始化可能成为一个计算任务代价的主要因素.我们随机生成了多个点查询,在 TPCH 数据集上(SF=10)对 Hadoop++进行了测试.图 3 是我们统计的 Hadoop++系统 Map Task 中各个阶段的执行时间.由于 Hadoop++对磁盘 I/O 进行了比较好的优化,因此数据加载阶段在整个 Map Task 中占比并不高,而 Map Task 的初始化在整个 Map Task 执行时间中比重非常高.

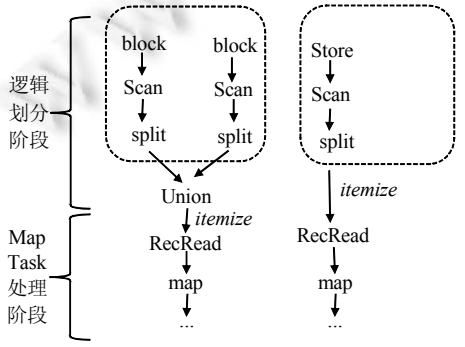


Fig.2 Data loading process in a Map Task

图 2 数据读取过程

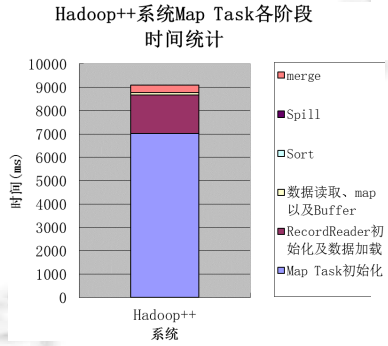


Fig.3 Time breakdown of a Map Task (Hadoop++)

图 3 Hadoop++系统 Map Task 各阶段时间统计

操作符的运算和查询有关,其数据查询优化方面的内容不在本文讨论范围.因此,对于数据加载阶段的优化可以采用减少数据读取的磁盘 I/O 以及减少 Map Task 的启动数目这两种方式.

2 HIndex 索引框架

Hadoop++,ORCFile 等技术主要通过减少数据访问的磁盘 I/O 提高查询效率,但是仍需要启动大量 Map Task 去处理不符合查询要求的 split.本节主要介绍我们提出的索引框架,从减少磁盘 I/O 和减少启动 Map Task 数目两个层次过滤数据.

HIndex 索引框架主要分为两层(如图 4 所示).

- 第 1 层是 split 层过滤,主要通过存储于内存中的统计信息表,在逻辑划分阶段过滤掉不满足查询条件的 split,减少数据读取产生的磁盘 I/O 和 Map Task 的启动数量.统计信息表存储于主节点的本地磁盘,系统启动时加载到内存中;

- 第 2 层是 split 内过滤,该层主要通过聚集索引和非聚集索引,只读取满足过滤条件的记录位置,避免不必要的数据读取产生额外的磁盘 I/O.索引存储于 HDFS 中,在处理查询时读取.

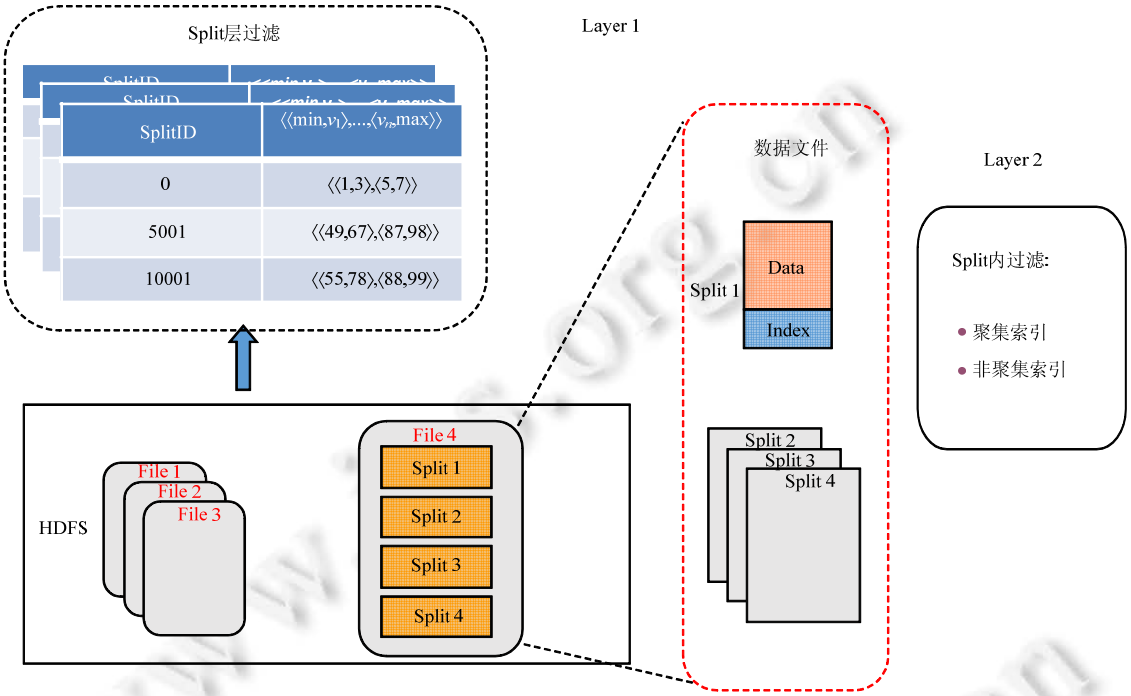


Fig.4 Framework of HIndex

图 4 HIndex 索引框架

3 split 层过滤

由基于 SOH 加载数据的分析可知,可通过减少 Map Task 的启动数目和减少数据读取的磁盘 I/O 来提高数据加载阶段的性能.本节主要介绍 split 层过滤,在内存的统计信息表过滤不必要的 split,从而减少数据读取的磁盘 I/O 和 Map Task 的启动数目,来达到过滤数据的目的.

3.1 统计信息表

我们针对于每个数据文件的每个索引属性建立一张统计信息表,该统计信息表的一条记录即表示该表文件其中一个 split 索引属性值分布情况的描述.统计信息表建立完成之后存于主节点的本地磁盘,当系统启动时,伴随 HDFS 的 FSimage 镜像文件一起从本地磁盘加载到内存中.统计信息表结构形为

$$SplitID|\langle\langle v_1,v_2\rangle,...,\langle v_{m-1},v_m\rangle\rangle.$$

上述 SplitID 即为对应的 split 的 ID.$\langle\langle v_1,v_2\rangle,...,\langle v_{m-1},v_m\rangle\rangle$为该 split 的数据区间,$v_1$ 为该 split 中对应的索引属性的最小值,v_m 为该 split 中对应的索引属性的最大值.我们通过一维的数据区间表示 split 索引属性值的分布,比如$\langle\langle 1,8\rangle,\langle 14,22\rangle,\langle 46,51\rangle\rangle$,表示该 split 中相应属性包含的值的范围是 1~8,14~22 和 46~51.若某个查询在该属性的条件是[30~40],我们很容易得出该 split 上不存在符合查询条件的数据,因此无须启动 Map Task 加载这个 split.

显然,区间越多,越能精确描述属性值的分布,精准地实现过滤.然而,统计信息表是加载到内存中的,因此不能占用太多的空间.我们用一个常量 k 来限制一个 split 的索引属性的划分区间数,k 的大小由应用决定,比如服务器的内存大小、所有数据 split 的数量、split 数据的分布等.

下面我们讨论:给定整数 k 以及包含在 split 中某索引属性的值如何确定划分区间,以最大化过滤数据的能力.

问题描述:将 split 索引字段中所有不重复值看作集合 $S=\{v_1,...,v_n\}$,给定正整数 k,找到 k 个数据区间 R,满足下面的要求:

$$R=\{[a_1,b_1],...,[a_k,b_k]\}(a_1\leqslant b_1\leqslant...\leqslant a_k\leqslant b_k)$$

$$\text{Minimize}\sum_{i=1}^{k}b_i-a_i$$

$$\text{s.t. }\forall x\ x\notin R\rightarrow x\notin S$$

我们的目标是使得这k个区间的长度之和尽可能小,这样可以最大化过滤数据的效果.算法 1 是我们给出的启发式的算法(算法思想来源于文献[6]).

算法 1. 计算数据区间算法.

Input: *values*, *k* //*values* is a sorted array; *k* is the number of intervals;

Output: *k*′ (*k*′≤*k*) data regions.

1. *result*←empty array;
2. *n*←|*values*|;
3. **if** (*n*≤*k*+1)
4. **return** *values*;
5. **end**
6. compute the *gap* between each value and its previous one;
   //a *gap* is a data structure including *len*, its length, and [*a*,*b*], its range.
7. build a max heap *H* to store all the gaps according to the length of each *gap*;
8. **for** *i*=1 to *k*−1 **do**
9. *G*[*k*−*i*−1]←extract the root element from *H*;
10. heapify-down *H*;
11. **end**
12. *result*[0]←*values*[0];
13. **for** *i*=1 to *k*−1 **do**
14. *result*[2**i*−1]←*G*[*i*−1].*a*;
15. *result*[2**i*]←*G*[*i*−1].*b*
16. **end**
17. *result*[2**k*−1]←*values*[*n*−1];
18. **return** *result*;

算法 1 对已排序的 *values* 进行遍历,得到相邻两个 *value* 之间的距离 *gap*,在所有 *gap* 中,选出值最大的 *k*−1 个,其相对应的 *value* 值为 2*k*−2 个.然后,将这 2*k*−2 个 *value* 值与 *values* 中最小值和最大值组合成最终的 *k* 个数据区间作为结果返回.算法 1 中根据所有 gap 的长度建立大顶堆(步骤 7),然后抽取 *k*−1 次堆顶元素,并重构堆(步骤 8~步骤 11),从而得到长度最大的 *k*−1 个 *gap*.建堆的平均时间复杂度为 $O(n)$[11],每次抽取堆顶并重构堆的时间复杂度为 $O(\log n)$.因此,总耗时为 $O(n+k\times\log n)$.

3.2 查询处理过程

本节主要介绍 Split 层过滤的查询过程.用户查询时,在逻辑划分阶段调用相应 API 接口,从内存中获取到每个 split 的统计信息,根据每个 split 的统计信息,提前过滤掉不满足查询条件的 split.不满足查询条件的 split 将不再启动 Map Task 处理,也不会产生磁盘 I/O.查询具体过程如下:

1) 用户通过 Client 向主节点提交查询;

2) 主节点在逻辑划分阶段对表文件进行逻辑划分,划分为多个 split;
3) 针对每一个 split,调用相应 API 查询内存中的统计信息表,获取每个 split 的数据区间;
4) 根据获取的 split 数据区间以及查询条件,判断该 split 是否满足查询条件:若不满足,则直接丢弃;
5) 若满足,则由后续操作符继续进行处理,直至查询结束.

4 split 内过滤

split 内过滤主要是从减少数据读取产生的磁盘 I/O 的角度进行优化. split 内过滤主要通过对 split 建立索引,在查询执行阶段,通过查询索引只读取满足查询条件的数据,减少不必要的数据扫描产生的磁盘 I/O.split 内过滤主要包含两个部分:聚集索引和非聚集索引.

4.1 聚集索引

聚集索引主要是借鉴了 Hadoop++的技术.简述如下.

存于 HDFS 上的表文件在进行处理时会被划分为一个个 split,将每个 split 中的记录按索引属性进行排序,然后针对索引属性建立聚集索引,将聚集索引存放于数据之后存于 HDFS 中.其结构如图 5 所示.

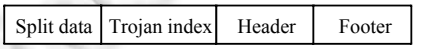


Fig.5 Structure of clustered index
图 5 聚集索引结构

Split Data 为该 split 的数据,Trojan Index 为针对此 split 的索引属性建立的聚集索引,其存放于数据之后.Header 存放索引的一些元信息,比如索引大小等.Footer 存放 split 的大小,主要用于重新划分新的 split,将原来的数据和生成的索引划分为一个新的 split.

如图 6 所示,Header 包含以下 5 个字段:DataSize 为该 split 数据的大小,IndexSize 为索引本身大小,Max 为索引属性在该 split 的最大值;Min 为索引属性在该 split 的最小值,RecordNum 为该 split 的 record 数量.Footer 中,SplitSize 和 FooterSize 分别表示 split 的大小和 Footer 的大小.

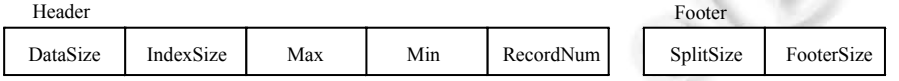


Fig.6 Structures of Header and Footer
图 6 Header 和 Footer 内部结构

- 建立索引

聚集索引可以通过 SOH 本身的计算框架建立,比如 MapReduce,Spark 等.这里介绍聚集索引的建立过程,此处以 MapReduce 为例.

1) 针对存放于 HDFS 的表文件,Map 过程会将其划分多个 split.split 中的每个键值对通过 *map* 函数进行处理,处理前⟨*key*,*value*⟩为{*pos*,*record*},经过 *map* 函数处理后为{*SplitID*+*a*,*record*},此处 *pos* 为该行在 split 中的偏移量,*a* 为索引属性的值.以图 7 中第 1 个数据记录为例,其 *map* 函数处理前的⟨*key*,*value*⟩为{0,{3223,1212,2300}},经过 *map* 处理之后,转化为{1+3223,{3223,1212,2300}};
2) 通过自定义函数 *Partitioner*,按 SplitID 进行划分,确保相同 Split 的数据交给同一个 Reducer 进行处理;
3) Reducer 处理阶段按照 SplitID 进行分组,并对于每个 split 按照 *a* 进行排序.如图 7 中按照 *Order_ID* 将该 split 的记录进行排序;
4) 针对 *a* 属性生成聚集索引.如图 7 所示,索引为 Trojan Index 记录了 *order_id* 的值和对应的偏移量.Header 记录数据和索引的元信息,Footer 记录新 split 的大小.

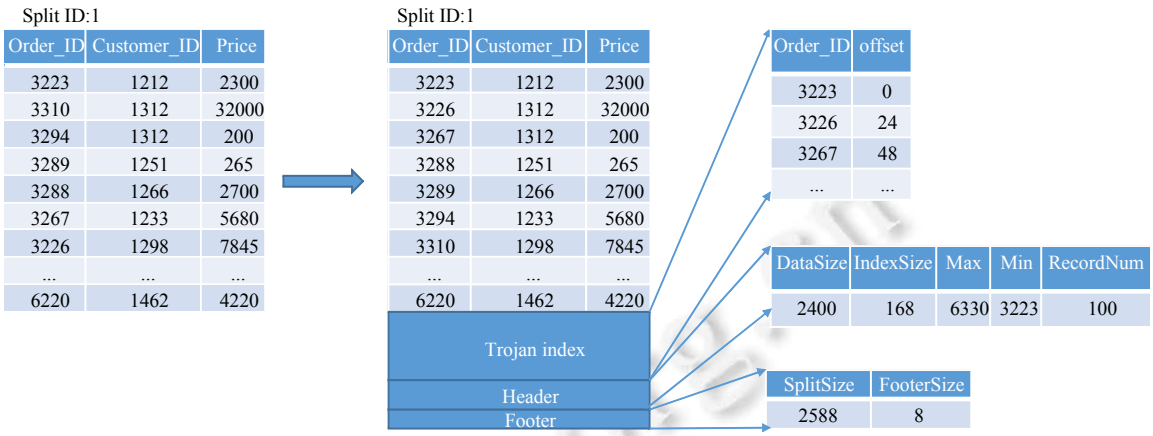


Fig.7 An example of constructing an index

图 7 建立索引过程示例

索引建立时间取决于所依赖的计算框架,但是因为本文主要针对于交互式查询,一般会将索引提前建好,因此并不影响查询的性能.由于 HDFS 数据“一次写入,多次读取”的特点,对于索引维护的需求并不是很大,因此,Hadoop++没有实现索引维护.

查询过程以 MapReduce 计算框架为例,其他计算框架同理,不再一一说明.

1) MapReduce 首先读取文件末尾处最后一个 Footer 字段,通过读取 splitsize 字段获取该 split 的大小,然后将该 split 划分出来;然后,读取倒数第 2 个 Footer,划分倒数第 2 个 split;依此类推;
2) 每一个新的 split 会交给一个从节点处理,读取 header 获取索引的元信息,比如索引大小等;
3) 读取索引,通过索引确定满足条件的记录的偏移量,查询并读取满足查询条件的记录.

4.2 非聚集索引

Hadoop++只实现了聚集索引,而没有考虑非聚集索引.在实际应用中,我们很多时候需要实现非聚集索引,比如查询的过滤条件不只是涉及到数据表的一个属性,而是该表的多个属性均有过滤条件.因此,本文设计并实现了非聚集索引.一个表可以同时拥有一个聚集索引和多个非聚集索引,以支持不同的查询要求.

非聚集索引的结构如图 8 所示.

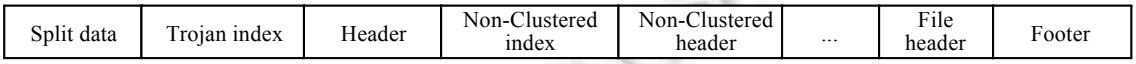


Fig.8 Structure of non-clustered index

图 8 非聚集索引结构

Split Data 为数据,Trojan Index 为聚集索引,Header 保存聚集索引的元信息,NonClustered Index 是非聚集索引,主要是保存未排序的属性的偏移量.NonClustered Header 保存非聚集索引的元信息.NonClustered Index 和 NonClustered Header 可以有多个,即可以同时建立多个非聚集索引.File Header 主要记录该 split 在哪些属性上建立了索引.Footer 记录该 split 的大小,用于划分 split.

- 建立索引

非聚集索引建立过程与聚集索引类似,但在数据排序等过程有所不同,具体过程如下.

1) split 中的每个键值对通过 *map* 函数进行处理,$\{pos,record\}\rightarrow\{SplitID+a+pos,record\}$;
2) 按照 SplitID 进行划分,确保相同 Split 的数据交给同一个 Reducer 进行处理;
3) Reducer 处理阶段,按照 SplitID 进行分组,并对于每个 split 按照 *pos* 进行排序.确保数据建立索引之前

和建立索引之后的相对顺序不变;

4) 针对属性 a 建立非聚集索引.

- 查询过程

非聚集索引查询过程与聚集索引有较大的不同:因为聚集索引的 split 是按照索引属性排序的,因此对于范围查询只需判断查询条件与 split 索引属性值范围的重叠范围,然后从头到尾扫描即可;而数据对非聚集索引属性是无序的,因此不能采用聚集索引的此种方法.

我们根据非聚集索引的特点和 SOH 数据扫描的特点制定了非聚集索引的扫描策略,我们假设 split 值的范围是[c,d],查询条件的查询范围是[a,b],这里有 6 种情况.

(1) 当 $c\leqslant a\leqslant d$ and $b\geqslant d$ 时,加载索引,扫描起始位置为[a,d]区间中所有值的偏移量最小值,终止位置为[a,d]中区间所有值的偏移量最大值;

(2) 当 $c\leqslant a\leqslant d$ and $c\leqslant b\leqslant d$ 时,加载索引,扫描起始位置为[a,b]区间中所有值的偏移量最小值,终止位置为[a,b]中区间所有值的偏移量最大值;

(3) 当 $a=b$ 时,加载索引,扫描起始位置为值 a 的偏移量最小值,终止位置为值 a 的偏移量最大值;

(4) 当 $a\leqslant c$ and $c\leqslant b\leqslant d$ 时,加载索引,扫描起始位置为[c,b]区间中所有值的偏移量最小值,终止位置为[c,b]中区间所有值的偏移量最大值;

(5) 当 $a\leqslant c$ and $b\geqslant d$ 时,全表扫描;

(6) 当 a,b 不满足以上 5 种情况时,丢弃该 split.

非聚集索引的查询过程基于以上的扫描策略,具体查询过程如下.

1) 首先读取文件末尾 Footer 字段,划分 split.类似聚集索引的方法将所有的 split 划分出来;

2) 读取 File Header 字段,判断查询条件中的属性是否建立了索引,并确定非聚集索引属性的偏移量;

3) 读取 NonClustered Header,获取非聚集索引的元信息.通过 Header 中记录的该 split 值的范围和查询条件中的查询范围确定扫描策略;

4) 根据制定的扫描策略扫描数据.

非聚集索引有效性分析:首先,根据索引,可以避免扫描一些完全无关的 split;其次,当 split 范围和查询范围重叠时,我们可以缩小扫描区域,避免读取整个 split 的记录;最后,当某个属性存在重复值时,通过索引可以定位到符合点查询条件的所有记录的偏移量范围,避免全表扫描.非聚集索引解决了查询包含多个属性过滤条件的问题.后续通过实验验证了非聚集索引的效率,证明了其可用性与有效性.

5 实验分析

5.1 实验设定和数据集

实验是在 Openstack 平台上进行的,我们使用了 8 个节点用于实验,其中,1 个为主节点,7 个为从节点.每个节点拥有 2 个 CPU 和 2GB 的内存,操作系统为 CentOS 6.5.每个节点拥有 200GB 的硬盘容量.系统源代码采用的是 Java.在数据集选择上,我们采用了通用的标准数据集 TPC-H,该数据集是模拟订单和仓库管理随机产生的数据,本实验选用的数据量为 10G.Hadoop 版本为 2.5.2.

5.2 数据区间粒度对于查询性能的影响

本节主要介绍数据区间粒度(即第 3.1 节中讨论的 k 值大小)对于查询性能影响的实验.数据区间粒度直接影响统计信息表的大小和查询的效率,如何确定数据区间粒度,是一个非常重要的问题.本实验通过调整数据区间粒度来观察查询的时间,从而确定出比较优的数据区间粒度,为后续实验奠定基础.

由图 9 可知:对于 TPC-H 数据集,查询时间随着数据区间划分个数增长而逐渐减少,到数据区间为 160 时基本趋于平缓.因此对于 TPC-H 数据集来讲,数据区间个数取值为 160 时是比较合理的.对于不同的数据集,最优的数据区间粒度有所不同,我们可以针对不同的数据集对数据区间粒度进行调整得到最优的数据区间粒度.

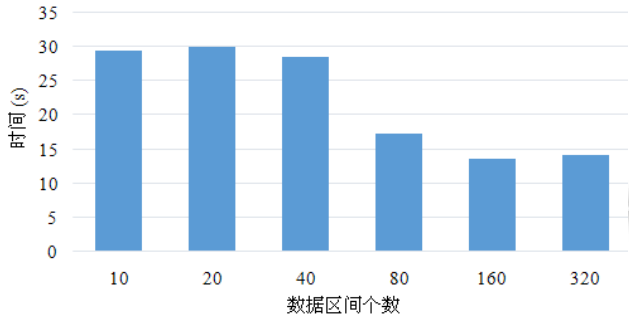


Fig.9 Influence of data ranges on query performance

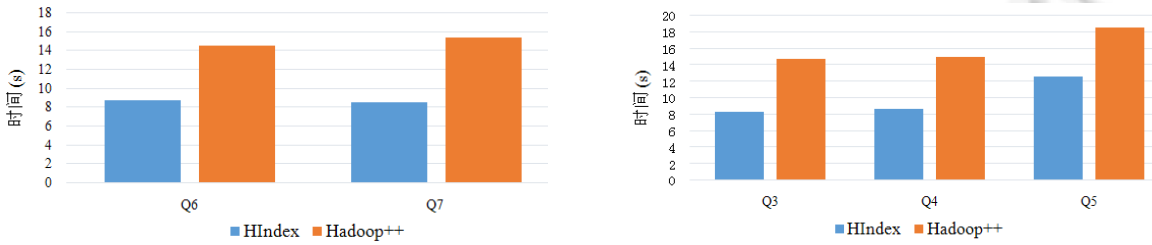
图 9 数据区间粒度对查询性能的影响

5.3 聚集索引性能测试

本节主要介绍聚集索引性能测试实验.在本实验中,我们的对比系统选择的是德国 Saarland 大学 Dittrich 等人提出的 Hadoop++系统.本实验从等值查询和短范围查询(查询区间小于相应的属性值域范围 5%)两个方面来测试.因为长范围查询涉及到大量的数据计算,基本已经接近于全表扫描,属于分析型查询的范畴,已经超出了本文的应用范围,此处不再做分析.

(1) 等值查询

图 10(a)是等值查询性能对比实验的结果.如图所示,HIndex 在 Q6,Q7 两个等值查询中性能比 Hadoop++有了明显的提升,查询性能分别提升了 40%和 44.2%.



(a) 等值查询性能对比实验 (b) 短范围查询性能对比实验

Fig.10 Performance comparison of clustered index

图 10 聚集索引性能对比

我们在表 1 中对比了 HIndex 和 Hadoop++两个系统 Map Task 启动的个数.Hadoop++两个查询启动 Map Task 数量均为 122,而 HIndex 系统整个查询只需要一个 Map Task 来处理,极大地优化了查询性能.表 1 的结果展示了 HIndex 在 split 层过滤的有效性.

Table 1 Number of Map Task for evaluating equality queries

表 1 等值查询启动 Map Task 个数对比表

Map Task 个数	Q6	Q7
HIndex	1	1
Hadoop++	122	122

(2) 短范围查询

图 10(b)展现的是短范围查询的实验结果.HIndex 相比于 Hadoop++在 3 个短范围查询 Q3,Q4,Q5 中查询速度分别提升了 43.5%,42.4%和 32.6%.

从表 2 中可以看出,HIndex 在 Q3,Q4,Q5 这 3 个查询中分别只需要启动 1,1,3 个 Map Task 去处理,优化效果

非常明显.

Table 2 Number of Map Task for evaluating short-range queries

表 2 短范围查询启动 Map Task 个数对比表

Map Task 个数	Q3	Q4	Q5
HIndex	1	1	3
Hadoop++	122	122	122

由于等值查询和短范围查询的结果较少,其满足查询条件的记录只存在于少量的 split 中,因此,大量的 split 是不需要启动 Map Task 去处理的.HIndex 可以完全避免启动 Map Task 所花费的开销,减少查询处理的代价,使得查询性能得到了提升.

5.4 非聚集索引性能测试实验

因为 Hadoop++系统并没有实现非聚集索引,因此,此实验直接与 Hadoop 进行对比.

(1) 等值查询

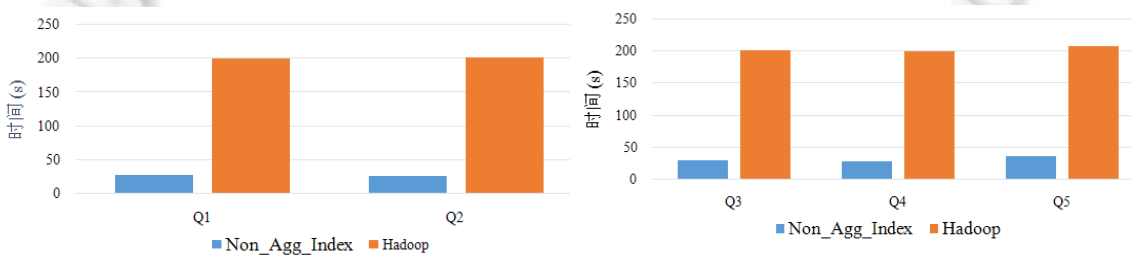
图 11(a)表明:非聚集索引在等值查询方面相较于 Hadoop 有很大优势,其在 Q1,Q2 性能分别提升了 7.22 倍和 7.5 倍.

(2) 短范围查询

在短范围查询方面,非聚集索引依然体现了明显的优势.如图 11(b)所示,HIndex 在 Q3,Q4,Q5 这 3 个短查询上查询速度相较于 Hadoop 分别提高了 7.12 倍、7.22 倍和 5.75 倍.

(3) 查询性能分析

对于等值查询和短范围查询,聚集索引显示出了巨大的优势.这是因为这两种的查询结果一般都比较少,如果此时还采用 Hadoop 全表扫描的方式,则会产生大量的磁盘 I/O,极大地增加了查询时间.通过非聚集索引,则可以提前过滤掉不需要的 split 和 split 内无用的数据,减少了磁盘 I/O,达到查询优化的目的.



(a) 等值查询 (b) 短范围查询

Fig.11 Performance comparison of non-clustered index

图 11 非聚集索引对比实验

6 结束语

本文针对基于 HDFS 的 SOH 系统在处理选择型查询或交互式查询时遇到的性能瓶颈提出了一种多层索引技术 HIndex.该索引包括 split 层和 split 内部两层过滤,实现了聚集索引和非聚集索引.实验结果显示,HIndex 可以有效地提高 SOH 系统查询处理效率.对于未来工作的展望:我们已开始在 Spark SQL 上集成本文所提出的索引技术.此外,我们将研究如何根据数据集规模和统计特征,自动确定数据区间粒度的选择(k).

References:

[1] Huai Y, Chauhan A, Gates A, Hagleitner G, Hanson EN, O'Malley O, Pandey J, Yuan Y, Lee RB, Zhang XD. Major technical advancements in apache hive. In: Proc. of the SIGMOD Conf. 2014. [doi: 10.1145/2588555.2595630]

[2] He YQ, Lee RB, Huai Y, Shao Z, Jain N, Zhang XD, Xu ZW. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In: Proc. of the ICDE. 2011. [doi: 10.1109/ICDE.2011.5767933]

[3] Melnik S, Gubarev A, Long JJ, Romer G, Shivakumar S, Tolton M, Vassilakis T. Dremel: Interactive analysis of Web-scale datasets. Proceedings of the VLDB Endowment, 2010,3(1):330−339.

[4] Dittrich J, Quiane-Ruiz JA, Jindal A, Kargin Y, Setty V, Schad J. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). Proceedings of the VLDB Endowment, 2010,3(1):518−529.

[5] Richter S, Quiané-Ruiz JA, Schuh S, Dittrich J. Towards zero-overhead static and adaptive indexing in Hadoop. In: Proc. of the VLDB. 2014. [doi: 10.1007/s00778-013-0332-z]

[6] Eltabakh MY, Özcan F, Sismanis Y, Haas PJ, Pirahesh H, Vondrak J. Eagle-Eyed elephant: Split-Oriented indexing in Hadoop. In: Proc. of the EDBT. 2013. 89−100. [doi: 10.1145/2452376.2452388]

[7] Gankidi VR, Teletia N, Patel JM, Halverson A, De Witt DJ. Indexing HDFS data in PDW: Splitting the data from the index. Proceedings of the VLDB Endowment, 2014,7(13):1520−1528. [doi: 10.14778/2733004.2733023]

[8] Lu P, Chen G, Ooi BC, Vo HT, Wu S. ScalaGiST: Scalable generalized search trees for MapReduce systems. Proceedings of the VLDB Endowment, 2014,7(14):1797−1808. [doi: 10.14778/2733085.2733087]

[9] Xin RS, Rosen J, Zaharia M, Franklin MJ, Shenker S, Stoica I. Shark: SQL and rich analytics at scale. In: Proc. of the SIGMOD Conf. 2013. 13−24. [doi: 10.1145/2463676.2465288]

[10] Chen GJ, Wiener JL, Iyer S, Jaiswal A, Simha RLN, Wang W, Wilfong K, Williamson T, Yilmaz S. Realtime data processing at facebook. In: Proc. of the SIGMOD Conf. 2016. [doi: 10.1145/2882903.2904441]

[11] Hayward R. Average case analysis of heap building by repeated insertion. Journal of Algorithms, 1991,12:126−153. [doi: 10.1016/0196-6774(91)90027-V]

何龙(1988－),男,河北石家庄人,硕士,主要研究领域为大数据管理技术.

陈晋川(1978－),男,博士,副教授,主要研究领域为大数据管理技术,不确定性数据管理技术.

杜小勇(1963－),男,博士,教授,博士生导师,CCF 会士,主要研究领域为数据库系统,大数据管理,智能信息检索,知识工程.