软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(3):634–647 [doi: 10.13328/j.cnki.jos.005906]

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

面向关系数据库的智能索引调优方法*

邱 涛[1], 王 斌[1], 舒昭维[1], 赵智博[1], 宋子文[1], 钟延辉[2]

[1](东北大学 计算机科学与工程学院,辽宁 沈阳 110169)

[2](华为技术有限公司 成都研究所,四川 成都 610000)

通讯作者: 王斌, E-mail: binwang@mail.neu.edu.cn

摘 要: 数据库索引是关系数据库系统实现快速查询的有效方式之一.智能索引调优技术可以有效地对数据库实例进行索引调节,从而保持数据库高效的查询性能.现有的方法大多利用了数据库实例的查询日志,它们先从查询日志中得到候选索引,再利用人工设计的模型选择索引,从而调节索引.然而,从查询日志中产生出的候选索引可能并未实际存在于数据库实例中,因此导致这些方法不能有效地估计这类索引对于查询的优化效果.首先,设计并实现了一种面向关系数据库的智能索引调优系统;其次,提出了一种利用机器学习方法来构造索引的量化模型,根据该模型,可以准确地对索引的查询优化效果进行估计;接着设计了一种高效的最优索引选择算法,实现快速地从候选索引空间中选择满足给定大小约束的最优的索引组合;最后,通过实验测试不同场景下智能索引调优系统的调优性能.实验结果表明,所提出的技术可以在不同的场景下有效地对索引进行优化,从而实现数据库系统查询性能的提升.

关键词: 索引调优;机器学习;数据库索引;优化模型;关系数据库

中图法分类号: TP18

中文引用格式: 邱涛,王斌,舒昭维,赵智博,宋子文,钟延辉.面向关系数据库的智能索引调优方法.软件学报,2020,31(3):634–647. http://www.jos.org.cn/1000-9825/5906.htm

英文引用格式: Qiu T, Wang B, Shu ZW, Zhao ZB, Song ZW, Zhong YH. Intelligent index tuning approach for relational databases. Ruan Jian Xue Bao/Journal of Software, 2020,31(3):634–647 (in Chinese). http://www.jos.org.cn/1000-9825/5906.htm

Intelligent Index Tuning Approach for Relational Databases

QIU Tao[1], WANG Bin[1], SHU Zhao-Wei[1], ZHAO Zhi-Bo[1], SONG Zi-Wen[1], ZHONG Yan-Hui[2]

[1](School of Computer Science and Engineering, Northeastern University, Shenyang 110169, China)

[2](Chengdu Research Institute, Huawei Technology Co. Ltd., Chengdu 610000, China)

Abstract: Indexing is one of the most effective techniques for relational databases to achieve fast queryprocessing. The intelligent index tuning technique can effectively adjust the index of the database instance to obtain efficient query performance. Most of the existing methods utilize the query log to generate candidate indices, and then use the artificially designed models to select indices, thereby the indices are adjusted. However, the candidate indices generated from the query log may not exist in the database instance, so they cannot precisely estimate the effects of such indices on the query processing. This study first designs and implements an intelligent index tuning system for the relational database. Secondly, it proposes a learning-based method to model the effects of indices for query processing, accordingly, the query optimization effect of an index can be accurately estimated when selecting optimized indices. Then, an efficient

* 基金项目: 国家重点研发计划(2018YFB1700404); 国家自然科学基金(U1736104, 61572122, 61532021); 中央高校基本科研专项资金(N171602003); CCF-华为数据库创新研究计划(CCF-Huawei DBIR2019009B)

Foundation item: National Key Research and Development Program of China (2018YFB1700404); National Natural Science Foundation of China (U1736104, 61572122, 61532021); Fundamental Research Funds for the Central Universities (N171602003); CCF-Huawei Database System Innovation Research Plan (CCF- Huawei DBIR2019009B)

本文由人工智能赋能的数据管理、分析与系统专刊特约编辑李战怀教授、于戈教授和杨晓春教授推荐.

收稿时间: 2019-07-20; 修改时间: 2019-09-10; 采用时间: 2019-11-25; jos 在线出版时间: 2020-01-10

CNKI 网络优先出版: 2020-01-10 13:34:38, http://kns.cnki.net/kcms/detail/11.2560.TP.20200110.1334.007.html

optimal index selection algorithm is designed to select a set of indices with the maximal utility from candidate indices, which satisfy the space threshold. Finally, experiments are conducted to test the performance of the proposed system in different settings. The experimental results show that the proposed technique can effectively adjust the index and achieve a significant improvement in query performance for a relational database.

Key words: index tuning; machine learning; database index; optimization model; relational database

数据库索引是数据库系统中一种排序的数据结构,以协助快速查询、更新数据库表中的数据[1,2].合理地设计数据库索引,可以有效提升数据库系统的检索速度.构建数据库索引虽可以提升数据库管理系统的查询性能,但同时也存在额外的磁盘开销与索引维护代价[3,4](例如更新查询引起的索引更新).因此,设计索引时需充分考虑应用的具体需求与数据分布特点,针对具体的实例添加相应的索引结构,从而提升数据库系统的查询性能.

当前,主流的索引设计方式是人工进行设计,由数据库管理员(DBA)或程序开发人员完成数据库实例的索引设计[2].这种设计方法存在如下两方面的限制:(1) 设计人员需要了解具体的应用需求(查询特征与分布等),并且熟悉所使用的数据库管理系统的查询优化策略与索引调用机制;(2) 当应用的查询特征与分布发生变化时,既有的索引结构无法及时调整,以保证数据库系统高效的查询性能.由于人工设计索引存在的这些限制,使得实际应用中会存在大量的索引设计不合理的情况,极大地影响了数据库系统的查询性能,同时也增加了服务器硬件资源的开销.

智能索引调优技术可以解决人工设计索引存在的问题,该技术通过对应用的查询日志进行分析,动态地对数据库实例的索引进行调整,使其对与查询日志中具有相似特征的查询提供高效的查询性能.现有的调优技术可以分为基于规则和基于代价模型的两类.基于规则的调优技术利用特定的规则(例如满足出现频率的字段组合[5]、满足索引调用机制的字段组合)从查询日志中产生出推荐的索引集合,如果推荐索引未存在于数据库实例中,则直接在实例中建立该索引.这类方法仅考虑了利用索引能带来的查询优化效果,未考虑维护索引所需的代价.尤其对于当前常见的混合事务/分析处理(HTAP)的应用,这类方法无法有效地调节索引.基于代价模型的调优技术则进一步考虑了维护索引所需的代价,通过设计相应的收益-代价模型,选择查询优化收益大于维护代价的索引.然而,从查询日志中产生的推荐索引可能并未实际存在于数据库实例中,现有的基于代价模型的方法不能准确地估计这类索引的查询优化效果与维护代价;同时,该类方法未考虑数据库实例中现有索引与数据分布对于查询性能的影响,致使其无法有效的进行索引调节.

另一方面,机器学习已经广泛地应用于各个领域.现有的技术已经利用了机器学习的方法来实现数据库的查询优化.例如,相比于传统的利用人工设计的模型,通过机器学习的方法可以更加准确地估计各种场景下的查询执行时间与资源消耗[6−9].本文针对上述基于模型的索引调优技术所存在的问题,提出了利用了机器学习的方法来预测不同的索引对于用户查询的查询优化效果.

本文首先设计并实现了一个面向关系数据库的智能索引调优系统,该系统可以直接应用于不同的关系数据库的实例上;其次,提出了一种利用机器学习的方法来构造对索引进行有效性量化的模型,根据该模型,可以准确地对索引的查询优化效果进行估计;接着设计了一种高效的最优索引选择算法,实现快速地从候选索引空间中选择最优的索引组合;最后,通过实验测试不同场景下智能索引调优系统的调优性能.实验结果表明:本文提出的技术可以在不同的场景下有效地对索引进行优化调节,从而实现数据库系统高效的查询性能.

1 相关工作

1.1 索引构建与优化方法

目前,最典型的索引设计与调节方法是由设计人员以离线的方式完成的.设计人员分析相应应用的具体查询业务,对于业务中常用的读查询(select 查询),对其查询条件所包含的字段建立二级索引,从而提升数据库系统的整体查询性能[1,2,4].然而,通过离线方式一次性构建的索引不能对所有的查询都具有查询优化效果,尤其对于混合事务/分析处理(HTAP)的查询需求.为了保证索引的有效性,设计人员需要长期地根据查询日志对数据库索

引进行调优.显然,由设计人员以离线的方式索引构建与调整需要很高的代价.

一些研究工作提出了自动索引调优的方法,该类方法避免了索引构建与调整过程中人工的参与,并且能够自动地根据应用查询日志来调整数据库索引.现有的自动索引调优技术可以分为基于规则[5]与基于代价模型这两类技术.MISA[5]利用的规则是构建索引的字段组合需要在查询日志中满足一定的出现频率,它利用频繁项挖掘算法 Apriori 找到查询日志中满足频率阈值的字段组合,然后在采样得到的数据库实例上再利用数据库优化器来进行筛选,并在最终选择的字段组合上建立二级索引.SOAR 是小米公司开发的一个用于 SQL 智能优化与改写的开源工具,同时也提供了索引调优的功能.SOAR 在通过查询日志进行索引调优时,使用的规则是选择满足索引调用机制[10]的字段组合建立索引.基于规则的技术没有考虑写查询(insert,update 与 delete 查询)引起的索引维护的代价,对于不同应用场景的索引调优需求,该类方法无法进行有效的索引调优.

基于代价模型的调优技术通过设计相应的收益-代价模型来选择索引,根据查询日志中包含的读/写查询,综合考虑了索引带来的查询优化效果与索引维护代价[11−14].该类方法的基本思路为:先通过查询日志产生出候选索引(字段组合),然后利用设计的模型对索引进行查询优化效果评估与维护代价评估.为了计算索引的查询优化效果,现有方法都是通过调用数据库系统的查询优化器来实现的(对于给定的查询,利用优化器比较索引存在与不存在这两种情况下该查询的查询代价).然而实际的数据库系统中,仅有 SQL Server 能通过 What-If 分析工具来实现类似的查询代价比较[11],极大地限制了方法的可用性.另一方面,现有方法都未考虑数据库实例中现有索引与数据分布对于查询性能的影响.

此外,还有一些工作研究了减小索引优化时添加索引对于数据库系统性能的影响[15−18].该类技术利用了增量索引构建的方法,其优先对某些数据表中的记录构建(partially-built)索引,并设计相应的利用 partially- built 索引的数据检索算法,使得 DBMS 可以利用部分建立好的索引来优化查询性能.这类方法与数据库系统的耦合度高,需要在特定的数据库系统上修改内核模块,并且大部分方法都是基于 NoSQL 数据库的技术,无法直接应用在关系数据库上.

1.2 利用机器学习的数据库查询优化方法

现有一些工作利用机器学习的方法来估计数据库查询的代价(查询时间).文献[6]采用统计学习的方法来估计 XML 查询的查询代价,其利用查询的操作符级别的特征,并使用一种基于变换回归(transform regression)的机器学习模型.文献[7]采用统计学习的技术来预测当存在多查询并发执行时,一组查询集合的完成查询所需的时间.文献[8]所提出的方法则用于估计单个查询的查询时间,该方法考虑了查询的查询模板中的特征与查询中操作符上的特征,并利用一种混合模型来同时使用这两类特征.文献[9]不仅可以预测查询的完成时间,还可以用于预估查询对于硬件资源的消耗(如 I/O 次数).该方法使用了与文献[8]类似的特征,并使用了一种基于回归树(regression tree)与尺度函数(scaling function)的混合模型.

2 智能索引调优系统框架

给定一个关系数据库的实例,智能索引调优系统利用该数据库实例上的一段时间窗口内的查询日志进行索引调优.调优系统包含了 3 个模块,分别为查询分析模块、候选索引生成模块与索引选择模块.图 1 所示为智能索引调优系统的总体框架.

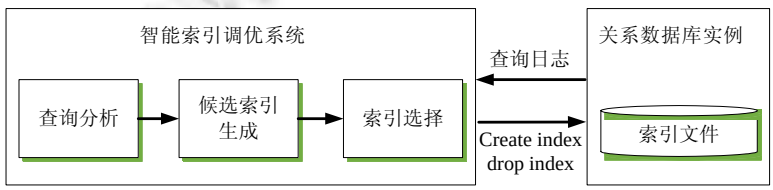


Fig.1 Framework of the intelligent index tuning system

图 1 智能索引调优系统的总体框架

调优系统最终生成的索引调节语句(创建索引/删除索引)会作用于数据库实例上.由于数据库实例的主键索引具有唯一性,修改主键索引可能会导致数据记录无法通过主键字段进行唯一标识.因此,调优系统仅针对数据库的二级索引(辅助索引)进行优化.图 2 所示为调优系统的各个模块的详细功能.

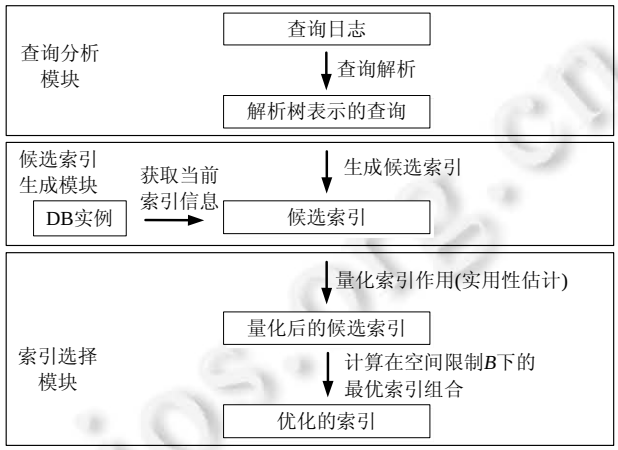


Fig.2 Functions ofdifferent modules in the intelligent index tuning system

图 2 智能索引调优系统的各个模块的详细功能

(1) 查询分析模块

按照查询对文件的操作方式分为读查询与写查询,本文考虑的读查询指仅包含 Select 操作的查询,包含 Insert/Delete/Update 操作的查询则为写查询.查询分析模块对查询日志中的每一条查询进行解析,解析过程包含词法分析与语法分析两个步骤,调优系统采用开源的词法分析器 Flex(GNU flex. free software foundation. https://www.gnu.org/software/flex/)与语法分析器 Bison(GNU bison. free software foundation. https://www.gnu.org/software/bison/)来完成词法分析与语法分析.在进行词法/语法分析的时候,系统同时能检验解析的 SQL 查询是否符合查询语言的词法与语法规则.解析后的查询可以表示为一个解析树的形式,系统可以通过该结构获取到相应查询上的投影(select)、选择(where)与排序(order by)操作使用到的字段组合.图 3 的示例为利用解析树获取查询中字段组合((CNO,FNAME),(LNAME,CNO),FNAME).

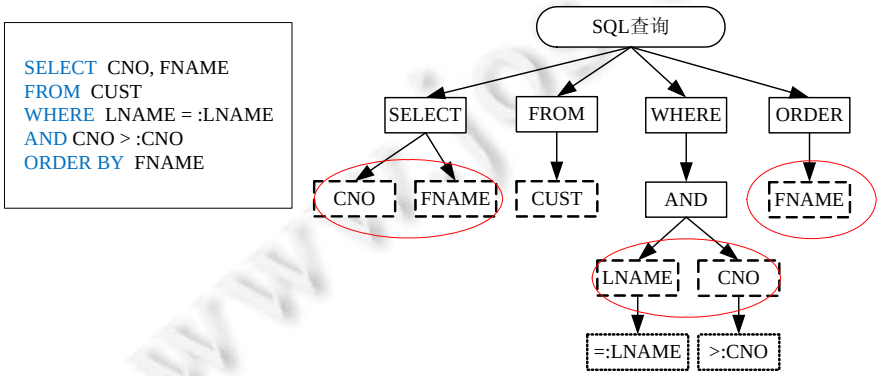


Fig.3 An example of computing field collections from the parsing tree of a SQL query

图 3 利用解析树获取查询中包含字段组合的示例

(2) 候选索引生成模块

对于查询分析模块产生的属于同一个数据表上的字段集合,其任意字段的组合都可以作为候选索引.然而,并非对所有的字段组合建立索引都可以提升查询的性能.例如,对于图 3 所示例子中获得的字段集合,在字段组

合(FNAME,LNAME)上建立索引不会对示例查询产生查询优化效果(即,对于该示例查询,查询优化器不会调用该索引).因此,为了获得有效的字段组合,本系统利用文献[10]提出的索引等级评价标准(即三星标准),对于查询记录中的每一条查询,都产生满足其任意星级标准的字段组合.其具体内容如下.

- 第 1 星索引:索引将相关的记录放到一起,即 Where 后面的等值谓词关联的列为组合索引的前缀;
- 第 2 星索引:索引中的数据顺序和查找中的排列顺序一致,即 Order by 中字段需被包含在组合索引中,且顺序一致;
- 第 3 星索引:索引中的列包含了查询中需要的全部列(覆盖索引).

例如,图 4 所示的示例中,IND_3 满足第 1 星索引,IND_4 满足了第 1 星与第 3 星索引,IND_5 则满足了第 2 星与第 3 星索引.另一方面,对于一个查询,只要该查询的 Where 条件中包含了一个索引的前缀字段(即前缀调用原则[10]),那么该查询就可以利用该索引进行查询优化.因此,一个查询的 Where 条件中包含字段的所有子集都被考虑为候选索引.图 4 所示的例子中,IND_1 与 IND_2 均为由 Where 条件中字段的子集产生的候选索引.

```
SELECT CNO, FNAME
FROM CUST
WHERE LNAME = :LNAME
AND CNO > :CNO
ORDER BY FNAME
```

候选索引

IND _1: (CNO)
IND _2: (LNAME)
IND _3: (LNAME,CNO) 第 1 星
IND _4: (LNAME,CNO,FNAME) 第 1 与第 3 星
IND _5: (LNAME,FNAME,CNO) 第 2 与第 3 星

Fig.4 An example of producing candidate indice

图 4 产生候选索引的示例

虽然不同的数据库系统或者不同的数据库系统版本在系统实现上会存在差异,但上述介绍的索引调用机制对于不同的关系数据库系统是一致的.

智能索引调优系统可以利用不同时段的查询日志连续地对数据库实例进行索引优化,因此,在进行一次索引优化之前,数据库实例可能会包含已经建立的二级索引,这种实际存在于数据库实例中的索引称之为真实索引.上述方法通过查询日志产生的实际不存在与数据库实例中的候选索引(非真实索引)称为虚拟索引.如图 2 中候选索引生成模块所示,调优系统最终生成的候选索引中包含了两类索引,即通过数据库实例获取到的真实索引与通过查询日志获取到的虚拟索引.

(3) 索引选择模块

索引选择模块的功能为从候选索引中选择一组索引总大小不超过用户给定空间阈值 B 的索引集合.该模块包含了两个步骤:第 1 步,通过定义索引实用值对索引的作用进行量化,并设计相应的模型来估计候选索引的实用值(详见第 3 节);第 2 步,从候选索引集合中选择一组满足空间阈值 B 的具有最大实用值的索引集合,使得优化后的索引集合的查询优化效果最大化(详见第 4 节).

3 索引查询优化效果建模

由于不同的索引对于数据库系统的查询优化效果是不一样的,甚至存在索引建立后会导致数据库系统查询性能降低的情况(由于索引存在维护代价).为了实现索引的调节优化,需要对索引的查询优化效果进行量化.对于一个候选索引 I,I 的实用值代表了该索引对于数据库系统在某个时间段内的查询优化作用,表示为 $I.utility$.本节先介绍索引实用值的具体内容,然后再介绍一种利用机器学习模型来估计索引实用值的方法.

3.1 索引实用值

给定查询日志 L,索引 I 对于 L 可获得的实用值可通过公式(1)进行计算:

$$I.utility = \alpha \cdot \sum_{q \in L} Profit(I,q) - \beta \cdot I.build \tag{1}$$

其中,$Profit(I,q)$表示索引 I 对于一个查询 q 所能取得的查询优化效果,$I.build$ 表示构建索引 I 所需要的时间代价.$I.utility$ 的计算考虑了 I 对于日志 L 中所有查询的作用(本文假设数据库系统未来处理的查询与查询日志 L 具有

相同的特征,因此可利用 L 来计算索引的实用值.当未来处理的查询与查询日志 L 的特征不一致时,可通过查询预测方法先得到预测后的查询负载[19,20],再利用预测的查询负载进行索引调优),并且还有构建 I 本身的代价.α与β为两个取值范围为[0,1]的因子,用于调节构建 I 的代价对于索引实用性的影响.α/β越大,则说明系统越倾向于索引带来优化效果,而不是考虑构建 I 所需的代价.

候选索引中的真实索引由于已经存在于数据库实例中,其 *I.build* 则为 0.对于虚拟索引,*I.build* 则是通过索引大小来进行估计的,即 $I.build=I.size\cdot unit\_cost$,其中,*I.size* 表示索引 I 的大小(真实索引大小可通过数据库实例直接获取到;对于虚拟索引,可以通过现有的估计模型进行计算(space needed for keys[21]),*unit_cost* 为通过测试得到构建单位大小(1MB)索引所需的时间.

实际上会存在多种形式的指标来表示索引对于查询的优化效果 $Profit(I,q)$,本文采用了查询 q 在索引 I 存在与不存在这两种情况下的执行时间差值来表示 I 的优化效果,即 $Profit(I,q)=time(q|I)-time(q|\bar{I})$.然而,$Profit(I,q)$是无法直接计算得到的,因为在索引调优阶段,是无法将虚拟索引在数据库实例中建立为真实索引的.因此,为了解决该问题,文本利用了机器学习的方法来预测 $Profit(I,q)$.该方法的基本思路为:先离线地通过训练数据集训练出一个反映查询 q、索引 I 与查询优化效果 $Profit(I,q)$映射关系的模型,再通过该模型来在线地对于给定 I 与 q 估计出对应的 $Profit(I,q)$,从而避免上述的索引构建问题.

3.2 利用机器学习预测索引优化效果

与许多现有利用机器学习的方法一样,本文的方法也分为两个阶段:训练阶段和测试阶段,如图 5 所示.

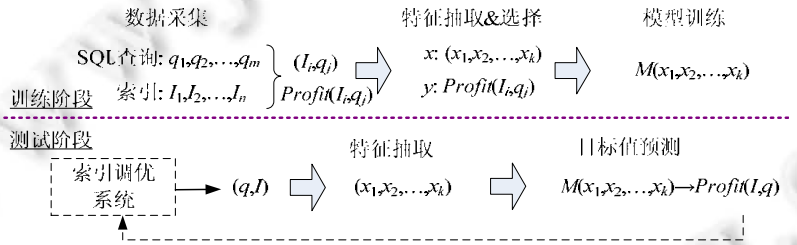


Fig.5 Phases of model training and testing

图 5 模型训练与测试阶段

训练阶段的目的为建立一个模型 M 来准确地反映出查询 q 与索引 I 的特征值与目标值 $Profit(I,q)$间的映射关系,即 $M(x_1,x_2,\ldots,x_k)\rightarrow Profit(I,q)$.训练阶段首先需要进行数据采集,对于训练数据(查询集合与索引集合)中的每一组(I,q),测试在没有任何索引情况下 q 的查询时间与仅存在索引 I 情况下的查询时间,该两种情况下的查询时间差值则为目标值 $Profit(I,q)$.由于不同的查询与索引对于 $Profit(I,q)$都是存在影响的,所以用于模型训练的特征属性需要同时从查询 q 与索引 I 中进行抽取,本文所使用的特征属性见表 1.

Table 1 Feature attributes selected from the query and index

表 1 查询与索引中选取的特征属性

名称	所属对象	描述
Q_type	查询 q	查询的类别(select,insert,update,delete)
Q_rows_num	查询 q	查询所涉及的表的记录数(多表则取平均值)
Q_est_rows	查询 q	查询影响的记录数的估计值
Q_est_cost	查询 q	查询的查询代价的估计值
Q_operator_num	查询 q	查询中包含的操作符个数
Q_fields_code	查询 q	查询中 Where 操作后所涉及的字段的编码
I_rows_num	索引 I	索引所在数据表的记录数
I_fields_num	索引 I	索引中包含的字段数目
I_fields_code	索引 I	索引所包含的字段的编码
IQ_invoking	查询 q 与索引 I	对于索引 I,是否满足前缀调用原则
IQ_updating	查询 q 与索引 I	是否会导致索引 I 的更新

表 1 中所示的特征属性都为查询编译阶段可获取到的属性值,特征属性可分为 3 类,分别从查询 q 与 I 中获取到的属性与通过 q 与 I 同时获取到的属性.其中,*Q_type*,*Q_rows_num*,*Q_operator_num*,*I_rows_num*,*I_fields_num* 均为调优系统在查询解析阶段可获取得到的属性值,*Q_est_rows* 与 *Q_est_cost* 为通过数据库系统查询优化器可获得的属性值(例如 MySQL 下的 EXPLAIN 命令).相比之下,*Q_fields_code* 与 *I_fields_code* 的获取则更为复杂,由于 q 与 I 所使用的字段信息都为字符串,为了得到可用作训练数据的数值型属性值,本文将数据库实例中的表中所有字段按照某一顺序进行排列,然后再将 q 与 I 中使用的字段按照该顺序进行 One-hot 编码[22],则得到 *Q_fields_code* 与 *I_fields_code*.最后,由于在查询解析阶段可以比较判断:i) q 是否满足 I 的前缀调用原则;ii) q 中更新的字段是否包含了 I 中的字段,因此可以获得 *IO_invoking* 与 *IO_updating* 这两个属性值.

训练阶段的最后一步为利用机器学习方法进行模型训练.本文使用了具有代表性的 3 种方法来训练模型.

- LR(linear regression)[23]:一种线性方法,数据使用线性预测函数来建模,并且未知的模型参数也是通过数据来估计.LR 的优点在于实现简单,建模速度快,并且可以有效地防止过拟合;但同时,LR 对异常值较敏感,并且线性模型本身表达能力差;
- GBDT(gradient boosting decision tree)[24]:一种由多棵决策树组成的用于回归分析的算法,在许多应用场景下,其具有很高的预测精度.GBDT 的优点在于可处理各种类型的数据(包括连续值与离散值),并且具有良好的非线性拟合能力,以及对超参数的鲁棒性;
- DNN(deep neural network)[25]:深度神经网络为当前最流行的方法,其本质是通过前面多层的隐藏网络学习抽象特征,在最后输出层使用上述抽象得到的特征完成最终的学习任务.这种方式得到的特征可以较好地降低问题的非线性程度,从而具有非常强的拟合能力.

本文在实验部分对上述方法进行了比较,并对比了训练出来的 3 种不同模型对索引优化的影响(详见第 5.2 节).在测试阶段,索引调优系统利用与训练阶段同样的方法,从候选索引 I 与查询 q 中提取出特征向量,然后调用训练好的预测模型来预测 I 对于 q 的查询优化效果 *Profit*(I,q),从而计算得到每个候选索引的实用值(公式(1)).

4 最优索引组合选择算法

4.1 最优索引选择问题

如上一节所述,对于候选索引集合 C 中的候选索引 I,其实用值为 *I.utility*、大小为 *I.size*.为了实现优化后的索引具有最优的查询优化效果,调优系统需要选择一组索引总大小满足用户给定阈值 B 且总的实用值最大的索引集合 S'(即 $\max\{\Sigma_{I\in S'}I.utility\},(\Sigma_{I\in S'}I.size)<B$).显然,该问题可以转化为经典的 0-1 背包问题[26].

设 X 为一组索引集合,y 为存储空间限制,$U(X,y)$表示在 y 的限制范围内选取 X 中索引可获得的最大的实用值.与 0-1 背包问题类似,$U(X,y)$可以通过如下递归方式进行计算,其中,$top(X)$为集合 X 中的第 1 个索引:

$$U(X,y)=\max\{U(X-top(X),y-top(X).size)+top(X).utility\} \tag{2}$$

对于经典的 0-1 背包问题,可以设计动态规划算法在时间 $O(m\cdot n)$内求得最优解,其中,m 为背包中物体的数目,n 为背包的容量.对于本节所需要解决的问题,直接利用该动态规划算法求解则会存在如下问题.

- 当调优系统所处理的数据库实例较大时,阈值 B 需设置较大的值.利用动态规划算法实现自底向上的求解子问题需要枚举每一种空间限制大小下的局部解,因此会导致很大的时间开销.例如,当阈值 B=1GB(空间大小粒度为 1MB)、候选索引数目为 1 000 时,计算的子问题数目为 1024×1000;
- 动态规划算法计算的子问题中,大部分都与最终的最优解无关.

因此,本文设计了自顶向下计算子问题的递归算法,该算法仅递归地计算与最终最优解有关的子问题,避免了动态规划算法导致的无关子问题的计算.同时,考虑到动态规划算法的优点在于可以避免重复子问题的计算(子问题结果复用),为了避免自顶向下的递归算法带来的重复子问题计算,本节提出了相应的技术解决递归算法中存在的重复子问题计算问题.

4.2 具有子问题复用的递归算法

在递归算法中实现子问题复用的基本思路为:对递归过程中计算过的子问题,利用哈希表存储其结果,在计算新的子问题时,先通过哈希表判断该子问题是否已经求解过.如哈希表中存在中间结果,则直接复用.

递归算法见算法 1,其设计思路与递归公式(2)是一致的.算法的输入为一组候选索引集合 X 与相应的大小阈值 y,输出为 X 中在满足 y 阈值下的最大的实用值与相应的索引集合.算法 1 首先判断当前递归处理的候选索引集合是否为空:如果为空,则表明输入的候选索引集合都已经处理完毕.

在 X 不为空时,算法会先对当前的 X 与 y 的组合求哈希值 h(该值用于标识当前子问题),然后判断哈希表 H(全局变量)中是否存在 h 对应的结果:如果存在,说明当前子问题已经计算过,则直接返回哈希表中存储的结果(算法 1 中第 2 行~第 4 行);如果 H 中不存在当前子问题的结果,则进一步递归地计算两类子问题,即 X 中舍弃了一个索引 I 与选择了索引 I 的情况(第 7 行与第 9 行).最后,比较这两类子问题所能取得的实用值大小,并且在哈希表 H 中记录相应的结果作为当前子问题的结果(第 11 行~第 14 行).

由于算法 1 利用哈希表避免了重复子问题的计算,其通过递归调用执行函数的次数最多为 $X\cdot y$ 次.此外,由于每次递归函数体中的执行操作复杂度为 $O(1)$,因此算法 1 的时间复杂度为 $O(X\cdot y)$.

算法 1. 最优索引选择算法(OptIDXSelect).

输入:候选索引集合 X,索引空间阈值 y;

输出:⟨选择索引的实用值 $U_{\max}$,选择的索引集合 $C_{\max}$⟩;

1. **If** $|X|\neq 0$ **then**
2. $h\leftarrow$***ComputeHash***(X,y); //计算(X,y)的哈希值
3. **If** $H[h]\neq NULL$ **then** //判断哈希表中是否存在结果
4. **Return** $H[h]$; //子问题结果复用
5. **Else**
6. $I\leftarrow X.$***pop***$(\cdot)$;
7. $\langle U_{\max1},C_{\max1}\rangle\leftarrow$***OptIDXSelect***$(X,y)$;
8. **If** $y\geqslant I.size$ **then** //判断当前空间大小对于 I 是否够大
9. $\langle U_{\max2},C_{\max2}\rangle\leftarrow$***OptIDXSelect***$(X,y-I.size)$;
10. $U_{\max2}+=I.utility$; $C_{\max2}.$***push***(I);
11. **If** $U_{\max1}\geqslant U_{\max2}$ **then**
12. $H[h]\leftarrow\langle U_{\max1},C_{\max1}\rangle$; **Return** $\langle U_{\max1},C_{\max1}\rangle$;
13. **Else**
14. $H[h]\leftarrow\langle U_{\max2},C_{\max2}\rangle$; **Return** $\langle U_{\max2},C_{\max2}\rangle$;
15. **Else**
16. **Return** $\langle 0,\varnothing\rangle$;

利用上述算法,将候选索引集合 C 与用户给定阈值 B 作为最初参数传入算法 1,即可计算得到具有最大实用值的最优索引集合 S'.然而通过实际的实验测试可以发现:对于相似的子问题(即 X 相同且 y 相近),其得到的子问题的解(索引组合)绝大多数的时候都是一样的.例如,y_1=991MB,y_2=992MB,对于相同的候选索引集合 X,$\{X,y_1\}$与$\{X,y_2\}$为两个不同的在递归时需计算的子问题,但$\{X,y_1\}$与$\{X,y_2\}$所求出的最优索引集合是一样的.之所以出现该现象,是因为少量的空间大小限制的变化对于求最优索引组合是没有影响的.

基于上述观察,本文进一步的提出了利用增大索引大小粒度来提升算法子问题复用能力的方法.设索引大小粒度为 $gran$(默认为 1MB),在算法 1 中计算(X,y)的哈希值时(第 2 行),将 y 除以 $gran$ 来提升子问题复用的能力,即 $ComputeHash(X,y/gran)$.通过该方法,相似的子问题(即 X 相同且 $y/gran$ 相同)的计算结果也可以进行复用.例如,在 $gran$ 取 10MB 时,$\{X,$991MB$\}$与$\{X,$992MB$\}$这两个子问题得到的哈希值是一样的,$\{X,$991MB$\}$下的最优索引组合计算完后,$\{X,$992MB$\}$可以直接利用$\{X,$991MB$\}$的计算结果.该方法虽然在小概率情况下会导致所求

的索引组合不是具有最大实用值的索引集合,但其可以提升索引调优系统的时间性能,尤其在候选索引多且索引空间阈值 B 较大时.在第 5 节,本文将通过实验进一步对该方法的效果进行阐述.

5 实 验

5.1 实验设置

本文在 MySQL 8.0 数据库系统上进行了实验测试,性能测试工具采用业界广泛使用的 TPCCMySQL 与 OLTPBench.测试的数据库实例为大小分别为 1GB 与 200MB 的两个 TPC-C 数据库实例(TPC-C 数据库实例采用的数据模型来自一个大型商品批发公司,其包含了 9 个数据表用以支持 5 类不同的交易事务).为了得到查询日志,首先将 MySQL 数据库的日志记录功能临时打开,然后分别利用 TPCCMySQL 与 OLTPBench 在测试实例上进行了性能测试,这样分别得到 TPCCMySQL 与 OLTPBench 产生的查询日志.性能测试阶段同样使用了 TPCCMySQL 与 OLTPBench,测试参数设置均为 c:4 r:60 l:600(即用户数为 4,预热时长 60s,测试时长 600s).在实验中,公式(1)中的参数α与β的取值分别为 0.6 与 0.4,表示调优系统在添加新的索引结构时,会同时考虑添加索引的带来的查询优化效果与索引构建的代价,但更侧重于添加索引的优化效果.

在默认条件下,TPCCMySQL 与 OLTPBench 会根据 TPC-C 实例信息来生成查询,并且生成查询的查询条件都可利用到对应表上的主键索引(在默认主键索引设置情况下,该工具产生的查询利用主键索引已经达到了最优的查询优化效果).因此,为了测试索引调优的效果,本文实验分别选择 3 种主键索引设置下的场景,从而使得测试工具生成的查询可通过建立二级索引进一步的提升查询性能.表 2 所示为 TPC-C 数据库实例上的 3 种不同的主键索引设置.

Table 2 Different settings for primary keys in the DB instance

表 2 数据库实例中不同的主键索引设置

设置名称	主键索引
PrimarySetting1	Item: i_id New_orders: no_w_id, no_d_id, no_o_id Order_line: ol_w_id, ol_d_id, ol_o_id, ol_number Stock: s_w_id, s_i_id Warehouse: w_id
PrimarySetting2	Customer: c_w_id, c_d_id, c_id District: d_w_id, d_id New_orders: no_w_id, no_d_id, no_o_id Orders: o_w_id, o_d_id, o_id Stock: s_w_id, s_i_id
PrimarySetting3	Customer: c_w_id, c_d_id, c_id Order_line: ol_w_id, ol_d_id, ol_o_id, ol_number Stock: s_w_id, s_i_id Warehouse: w_id

TPCCMySQL 工具产生的查询日志包含了 7 763 条查询记录,其中,不同类别的查询占比分别为:Select 61.67%,Delete 1.23%,Update 21.01%,Insert 16.09%(该比例由 TPCCMySQL 使用的查询事务比例所决定).此外,查询日志中,98.5%为单表查询,1.5%为多表查询.OLTPBench 工具则可通过变化事务的比例来产生不同查询类别占比的查询日志(详细信息见第 5.2 节中的第 3 个实验).本节所有实验均使用 TPCCMySQL 作为性能测试工具,性能测试指标为数据库系统的吞吐量(TPS,每秒执行的事务数).

用于测试的数据库实例部署在一台 ThinkServer 服务器上(配置为 Intel Xeon E3-1226 3.30GHz 处理器,16GB 内存,500GB SSD),索引调优系统部署在一台 Dell PC 上(配置为 IntelCore i7-8700 3.2GHz 处理器,8GB 内存,256GB SSD),调优系统通过远程连接数据库实例获取到实例信息与索引的调节.索引调优系统通过 C++语言编写与实现,估计模型通过 PyTorch 框架来编写,并将训练得到的模型存储到本地,再由调优程序进行调用.

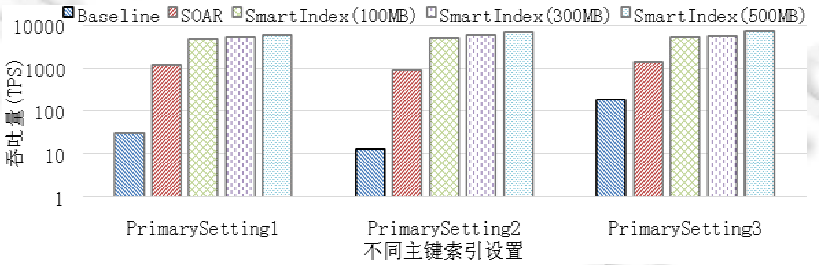
5.2 实验结果及分析

(1) 不同索引调优方法调优性能对比

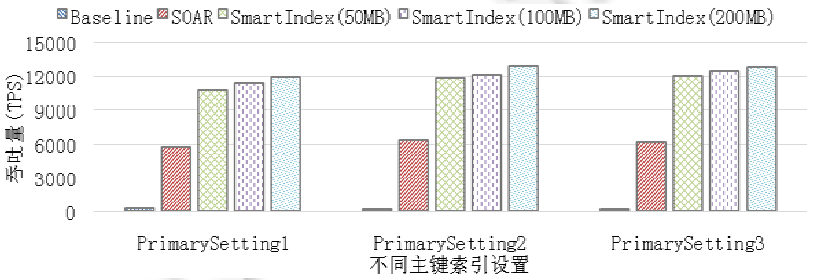
第 1 个实验对比了不同的索引调优方法的性能.对比的方法中,Baseline 为使用对应的主键索引设置的方法.由于现有的索引调优/推荐方法中,大部分都依赖查询优化器来实现索引作用的估计,无法应用在 MySQL 数据库系统上,因此,本文仅使用了开源的 SQL 优化器 SOAR(SQL optimizer and rewriter. Xiao Mi. https://github.com/xiaomi/soar)作为对比方法,其无需依赖查询优化器完成索引调优,并且对比方法不考虑索引大小的约束限制.SmartIndex 为本文所提出方法(默认情况下,SmartIndex 使用 DNN 来训练索引查询优化效果的估计模型,不同预测模型的对比测试结果如下一个实验所示).

图 6(a)所示为数据库实例大小为 1GB 时,不同方法的对比结果.SmartIndex 分别测试了索引空间阈值设置为 100MB,300MB 与 500MB 时的调优效果.通过实验结果可以发现:相较于 Baseline 方法,SOAR 与 SmartIndex 方法在索引调节后都取得了更好的查询性能,并且 SmartIndex 的索引调优效果要优于 SOAR.例如,在 PrimarySetting1 下,系统在利用 SOAR 方法调优后的吞吐量提升到了 1 200.相较之下,经过 SmartIndex(索引大小阈值 B 设置为 500MB)调优后,吞吐量则提升到了 5 993.在大小为 200MB 的数据库实例上也得到了相似的实验结果,如图 6(b)所示.例如,在 PrimarySetting3 下,经过 SmartIndex(阈值 B=500MB)调优后,数据库系统的吞吐量从 190 提升到了 12 778.由于图 6(b)中的实验所用数据库实例要小于图 6(a)实验中的实例,因此图 6(b)中实验所获得的吞吐量要明显高于图 6(a)中的实验结果.另一方面,通过比较 PrimarySetting1,PrimarySetting2 与 PrimarySetting3 下的实验结果可以发现,SmartIndex 在不同的主键索引设置下都具有很好的调优效果.

此外,图 6 所示实验还测试了不同索引大小阈值 B 对于索引调优的影响.实验结果显示:SmartIndex 在越大的索引大小阈值下,可以取得越好的查询性能.例如,在图 6(a)中的 PrimarySetting1 下,当阈值 B 的值从 100MB 增加到 500MB 时,系统的吞吐量从 4 820 增加到了 5 993.这是因为阈值 B 越大的情况下,系统可以建立更多有效的索引,从而更多的查询可以利用索引来提升查询性能.



(a) TPC-C 实例大小为 1GB 时的测试结果



(b) TPC-C 实例大小为 200MB 时的测试结果

Fig.6 Comparison on the tuning performance of different index tuning methods

图 6 不同调优方法的调优性能对比

(2) 索引优化作用估计模型测试

第 2 个实验测试了不同估计模型训练算法对于索引调优效果的影响.本实验所对比的模型训练方法为 LR,GDBT 与 DNN(如第 3.2 节所述).对于 LR 方法,实验使用了前向特征选择(forward feature selection)[27]的方法来

选择特征属性,从而使得 LR 训练的模型达到了最高的准确率.DNN 算法使用了 2 层隐藏层,并且使用 Adam 算法作为优化器.为了进行数据采集,实验利用 TPCCMySQL 产生了 2 000 条的查询日志集合,并且在 TPC-C 数据库实例上建立了 80 个不同的索引.对于一个查询 q 与索引 I,为了测试得到 $Profit(q,I)$,实验分别在不包含任何索引的数据表与仅包含 I 的同一数据表上进行了性能测试,然后求两种情况下的执行之间的差值.最终,实验采集得到 160 000 条具有真实目标值的数据,并将数据按照 7:3 的比例划分成训练数据与测试数据.

由于绝对误差的度量方式存在个别数据误差影响整体误差的问题,为了度量预测模型的准确性,本文使用了文献[19]中定义的平均相对错误率作为度量标准,公式(3)所示为平均相对错误率的计算方式:

$$Relative\_error=\frac{1}{|S_{test}|}\sum_{\langle q,I\rangle\in S_{test}}\frac{|Profit(q,I)_{actual}-Profit(q,I)_{estimate}|}{Profit(q,I)_{actual}} \tag{3}$$

首先,实验在测试数据上利用 3 种方法训练出来的模型进行了预测测试,实验结果见表 3 中第 1 行数据所示.DNN 的模型获得了最低的相对错误率,仅为 32.6%;LR 的模型的错误率最高,达到了 53.1%.实验进一步比较了不同模型用于索引调优时的效果,测试所用数据库实例大小为 1GB,SmartIndex 设置的索引大小阈值 B 为 300MB.实验结果显示:由于 DNN 的模型具有最高的预测准确率,因此使用该模型进行索引调优后获得了最大的吞吐量.此外,虽然 LR 与 GDBT 模型的错误率要明显高于 DNN 的错误率,但利用该模型取得的 *TPS* 差距却要小一些.例如,在 PrimarySetting1 下,利用 LR 与 DNN 的方法取得 *TPS* 最大差距也仅有 24.5%.出现该现象的原因是:不同的查询与索引利用同一预测模型后得到的目标值都会存在一定误差,但不同索引估计得到的实用值间的相对关系却是正确的.因此,利用本文所提出的索引选择方法(见第 4 节)仍能选择出有效的索引组合.

Table 3 Comparison for the models trained by different learning algorithms

表 3 不同算法训练得到的预测模型的性能比较

评测指标	LR	GDBT	DNN
Relative_error (%)	53.1	45.5	32.6
TPS(PrimarySetting1)	4 060	4 231	5 377
TPS(PrimarySetting2)	4 679	4 933	6 058
TPS(PrimarySetting3)	4 221	4 344	5 671

(3) 不同类型查询占比对调优性能影响测试

由于写查询(增删改)可能会导致索引的更新,因此不同的查询类型对于索引的查询优化效果会存在影响.为了测试不同查询集合对于参数调优的影响,第 3 个实验采用 OLTPBench 工具来产生查询日志并且进行性能测试(OLTPBench 可以通过调节事务比例来产生不同类别的查询).表 4 所示为 OLTPBench 产生的 3 组不同的查询日志,3 组查询日志中包含的 Select 查询占比分别为 85.12%,62.64%与 49.80%,其分别代表了读密集型查询、混合类型查询与写密集型查询.

Table 4 Three different query work load generated by OLTPBench

表 4 OLTPBench 产生的 3 组不同的查询日志

名称	推荐索引所用查询数	不同查询占比
Query Worload1	4 870	SEL: 85.12%, INS: 6.33%, UPD: 8.06%, DEL: 0.49%
Query Worload2	11 037	SEL: 62.64%, INS: 16.48%, UPD: 19.53%, DEL: 1.35%
Query Worload3	15 863	SEL: 49.80%, INS: 18.50%, UPD: 28.91%, DEL: 2.79%

图 7 所示为利用 SmartIndex 方法进行索引调优后(Baseline 为 PrimarySetting1),通过 OLTPBench 进行性能测试得到的实验结果(性能测试的查询类别比例与生成查询日志中的查询类别比例一致).随着写查询比例的增加,通过索引调优后的数据库系统的吞吐量是降低的.例如,SmartIndex(B=300MB)在 query workload1 与 query workload3 调优后得到的系统吞吐量分别为 1 585 与 518.出现这种现象的原因是:更多的写查询会增加索引的更新维护代价,从而降低了整体的吞吐量.此外,实验结果还显示:在写查询比例增加后,增大索引阈值所能提升的查询性能是降低的.这是因为在写查询比例高的时候,即使允许建立更多的索引,但由于索引更新维护代价的增加,SmartIndex 也无法选择出新的具有查询优化效果的索引.

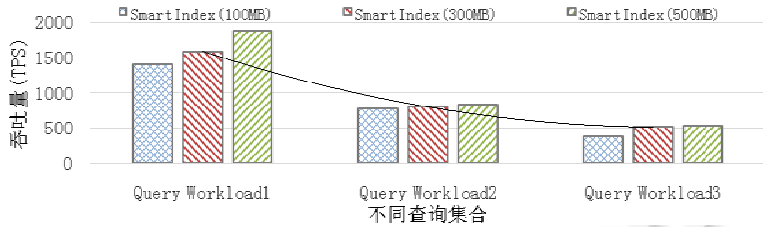


Fig.7 Impact of different query workloads for the effect of index tuning

图 7 不同类型的查询集合对于调优性能的影响

(4) 索引选择算法对比

本文最后一个实验测试了第 4 节提出的索引选择算法的效果.实验在 1GB 的 TPC-C 数据库实例上利用 TPCCMySQL 性能测试工具,测试对比了贪心算法 Greedy(优先选择单位实用值高的索引,即 *I.utility*/*I.size*)、动态规划算法 DP 与本文提出的 OptIDXSelect 算法,同时还对比了 OptIDXSelect 算法在索引大小粒度 *gran* 设置为 10MB 与 100MB 下的效果.

图 8 与图 9 所示分别为不同索引选择算法得到的索引调优效果与推荐索引所需的时间.

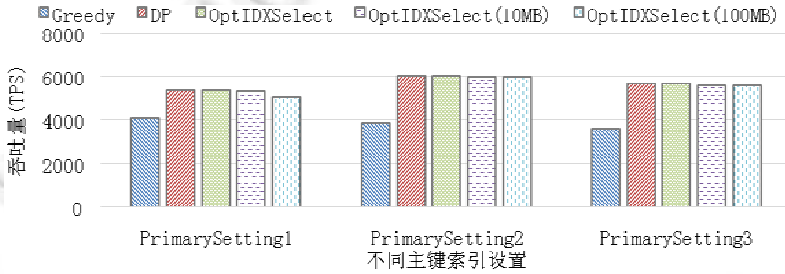


Fig.8 Comparison on the effect of index tuning for different index selection algorithms

图 8 不同索引选择算法的索引选择效果对比

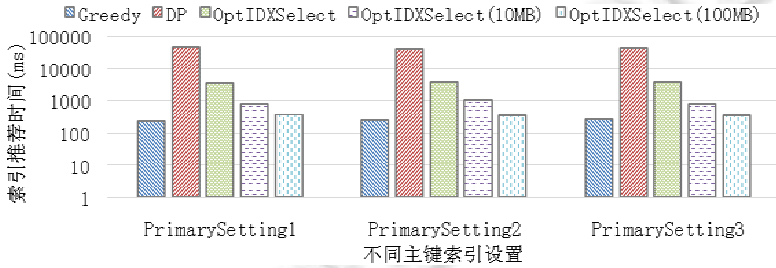


Fig.9 Comparison on the time of computing optimized indices for different index selection algorithms

图 9 不同索引选择算法完成索引推荐的时间对比

通过图 8 实验结果可以发现,DP 与 OptIDXSelect 所选择索引的优化效果要优于 Greedy 方法所选择的索引.例如,在 PrimarySetting1 下,Greedy,DP 与 OptIDXSelect 选择的索引得到的吞吐量分别为 4 113,5 377 与 5 377.同时需注意到:DP 与 OptIDXSelect 方法由于都是选择一组具有最大实用值的索引,因此它们得到的吞吐量是一致的.OptIDXSelect 算法在设置索引大小粒度 *gran* 后,所选择的索引获得的吞吐量有所下降.这是因为设置 *gran* 的目的是加快索引的选择,但同时不能保证选择的索引取得最大的实用值.

图 9 进一步反映了各种方法在选择索引上的时间效率.显然,由于 Greedy 使用了贪心策略,其选择索引花费的时间是最少的.比较 DP 与 OptIDXSelect 这两个可以取得最优解的方法,OptIDXSelect 的时间效率要明显优于

DP.OptIDXSelect(*gran*=10MB)相比于 OptIDXSelect 获得的吞吐量相差很小,但是其选择索引的效率却要远高于 OptIDXSelect 方法.例如,在 PrimarySetting1 下,Greedy,DP,OptIDXSelect,OptIDXSelect(*gran*=10MB)还有 OptIDXSelect (*gran*=100MB)用于选择索引花费的时间分别为 221ms,43792ms,3388ms,774ms 与 367ms.虽然当前实验测试不同算法差距最大的也仅为 43571ms,但在需要连续的通过查询日志进行索引调优,并且日志中包含大量查询的时候,选择高效的算法来完成索引选择仍非常重要.

6 总结与未来的工作

索引是数据库系统实现高效的查询性能的方式之一,智能地对数据库实例构建并调节索引,不仅能够保证高效的查询性能,还能有效地提高硬件资源利用率与减少人力成本的投入.本文研究了面向关系数据库的智能索引调优方法.首先,本文设计并实现了一个面向关系数据库的智能索引调优系统;在索引调节时,为了准确地估计索引对于查询的查询优化效果并选择正确的索引,本文设计了一种使用机器学习方法来对索引的查询优化效果建模的方法;为了最大化索引的调优效果,本文进一步地设计了一种高效的最优索引组合选择算法,并提出了优化技术来提升索引选择的效率;最后设计了一系列的实验对索引调优的各个环节进行了测试.结果表明:本文设计的系统可以在不同的场景下有效地对索引进行调优,从而提升数据库系统的查询性能.

利用智能技术优化数据库索引具有很大的研究空间,未来的工作中将从以下 3 个方面展开:首先,探索利用混合模型来提高索引查询优化效果的估计准确性;其次,由于同一查询可能利用上不同的索引来提升查询效率,通过考虑不同索引间的相互影响,进一步提升索引选择的质量;最后,进一步对比研究回归任务与分类任务对于索引优化效果估计的影响.

References:

[1] Silberschatz A, Korth HF, Sudarshan S. Database System Concepts. 4th ed., New York: McGraw-Hill, 1997.

[2] Schwartz B, Zaitsev P, Tkachenko V. High Performance MySQL: Optimization, Backups, and Replication. 3rd ed., O'Reilly Media, Inc., 2012.

[3] Zilio DC, Rao J, Lightstone S, *et al*. DB2 design advisor: Integrated automatic physical database design. In: Proc. of the 30th VLDB Conf. 2004. 1087−1097.

[4] Finkelstein S, Schkolnick M, Tiberio P. Physical database design for relational databases. ACM Trans. on Database Systems, 1988, 13(1):91−128.

[5] Ameri P, Meyer J, Streit A. On a new approach to the index selection problem using mining algorithms. In: Proc. of the 2015 IEEE Int'l Conf. on Big Data. IEEE, 2015. 2801−2810.

[6] Zhang N, Haas PJ, Josifovski V, *et al*. Statistical learning techniques for costing XML queries. In: Proc. of the 31st Int'l Conf. Very Large Data Bases. 2005. 289−300.

[7] Ahmad M, Duan S, Aboulnaga A, *et al*. Predicting completion times of batch query workloads using interaction-aware models and simulation. In: Proc. of the 14th Int'l Conf. on Extending Database Technology. ACM, 2011. 449−460.

[8] Akdere M, Çetintemel U, Riondato M, *et al*. Learning-Based query performance modeling and prediction. In: Proc. of the 28th Int'l Conf. on Data Engineering. IEEE, 2012. 390−401.

[9] Li J, König AC, Narasayya V, *et al*. Robust estimation of resource consumption for SQL queries using statistical techniques. Proc. of the VLDB Endowment, 2012,5(11):1555−1566.

[10] Lahdenmaki T, Leach M. Relational Database Index Design and the Optimizers: DB2, Oracle, SQL Server. John Wiley & Sons, 2005.

[11] Chaudhuri S, Narasayya VR. An efficient, cost-driven index selection tool for Microsoft SQL server. In: Proc. of the 23rd VLDB Conf. 1997. 146−155.

[12] Bruno N, Chaudhuri S. An online approach to physical design tuning. In: Proc. of the 23rd Int'l Conf. on Data Engineering. IEEE, 2007. 826−835.

[13] Schnaitter K, Abiteboul S, Milo T, *et al*. Colt: Continuous on-line tuning. In: Proc. of the 2006 ACM Int'l Conf. on Management of Data. ACM, 2006. 793−795.

[14] Sattler KU, Geist I, Schallehn E. Quiet: Continuous query-driven index tuning. In: Proc. of the 2003 VLDB Conf. 2003. 1129−1132.

[15] Petraki E, Idreos S, Manegold S. Holistic indexing in main-memory column-stores. In: Proc. of the 2015 ACM Int'l Conf. on Management of Data. ACM, 2015. 1153−1166.

[16] Idreos S, Kersten ML, Manegold S. Database cracking. In: Proc. of the Conf. on Innovative Data Systems Research, Vol.7. 2007. 68−78.

[17] Voigt H, Kissinger T, Lehner W. Smix: Self-managing indexes for dynamic workloads. In: Proc. of the 25th Int'l Conf. on Scientific and Statistical Database Management. ACM, 2013.

[18] Goel S, Langford J, Strehl AL. Predictive indexing for fast search. In: Proc. of the Advances in Neural Information Processing Systems. 2009. 505−512.

[19] Ma L, Van Aken D, Hefny A, *et al*. Query-based workload forecasting for self-driving database management systems. In: Proc. of the 2018 Int'l Conf. on Management of Data. ACM, 2018. 631−645.

[20] Kesarwani M, Kaul A, Singh G, *et al*. Collusion-resistant processing of SQL range predicates. Data Science and Engineering, 2018, 3(4):323−340.

[21] Oracle. MySQL 8.0 Reference Manual. MySQL Press, 2018.

[22] Harris D, Harris S. Digital Design and Computer Architecture. Morgan Kaufmann Publishers, 2010.

[23] Freedman DA. Statistical Models: Theory and Practice. Cambridge University Press, 2009.

[24] Chen T. Introduction to Boosted Trees. University of Washington Computer Science, 2014.

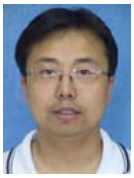
[25] Lecun Y, Bengio Y, Hinton G. Deep learning. Nature, 2015,521(7553):436.

[26] Martello S. Knapsack Problems: Algorithms and Computer Implementations. Wiley-InterscienceSeries in Discrete Mathematics and Optimization, 1990.

[27] Witten IH, Frank E, Hall MA, Pal CJ. Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann Publishers, 2016.

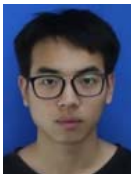
邱涛(1989－),男,江西萍乡人,博士,主要研究领域为文本数据管理,复杂查询优化处理.

赵智博(1996－),男,硕士生,主要研究领域为查询优化,索引优化.



王斌(1972－),男,博士,副教授,博士生导师,主要研究领域为查询优化处理,图数据管理,隐私保护.

宋子文(1992－),男,硕士生,主要研究领域为查询优化,索引优化.



舒昭维(1996－),男,硕士生,主要研究领域为查询优化,索引优化.

钟延辉(1984－),男,系统架构师,主要研究领域为数据库,分布式系统.