https://doi.org/10.1051/jnwpu/20213940909

基于分布式数据库的相关子查询优化

张晨煜[1], 刘文洁[1], 庞天泽[2], 岳艳涛[2]

(1.西北工业大学 计算机学院, 陕西 西安 710129; 2.交通银行, 上海 200120)

摘 要:子查询在数据库中的应用较为广泛,根据是否与父查询的表有依赖关系,可以将其分为相关子查询和非相关子查询。相关子查询需要先从父查询中取出一个元组后执行子查询,即需要反复对子查询的内容进行运算。这种策略的磁盘访问开销很大,在分布式数据库中,由于存在数据通信开销,在父查询元组过多时效率较低。针对该类子查询,在现有的优化查询策略的基础上,结合分布式数据库的特点,提出了通过子查询上拉为连接查询,消除子查询中冗余子句,消除聚集函数等方法实现的基于分布式数据库的子查询优化策略,并通过实验验证了所提优化策略的有效性。

关 键 词:分布式数据库;相关子查询优化;基于规则的优化

中图分类号:TP311.1 **文献标志码:**A **文章编号:**1000-2758(2021)04-0909-10

随着互联网的发展和数据量的不断激增,分布式数据库已经逐渐取代单机数据库,成为金融、互联网等行业应用的主流存储系统。

在当前的数据库使用中,读操作占了数据库操作的大多数,反映到 SQL 语句中即查询语句被经常使用。将一个由 SELECT-FROM-WHERE 组成的结构称为查询块,对于一个查询块作为另一个查询块的查询条件嵌套在其中的语句,称为子查询[1]。目前对于子查询的执行,如果是不依赖于父查询的非相关子查询,实现比较简单,只需先对子查询进行处理,将结果与父查询绑定后执行父查询,由内向外每个查询只需要执行一次。但对于依赖于父查询的相关子查询,需要先对父查询计算,对于父查询结果中的每一个元组,都需要将其重写到子查询中进行一次子查询的计算。这种执行策略使得相关子查询的执行会随着父查询元组数的增多而增长,对于嵌套层次更多的复杂相关子查询,时间开销将会是指数级增加。同时,在分布式数据库环境中,这种执行策略还会增加数据通信的时间开销。

子查询为 SQL 查询提供了更为丰富的表达能力,在数据库使用当中,大多数查询语句都含有子查询,比如在 TPC-H 基准测试中就含有近乎一半的子查询语句。因此,对于相关子查询进行性能优化,对于分布式数据库的高效执行有着重要的作用。

现有的查询优化主要是基于 2 个方面进行,即基于规则的查询优化和基于代价的查询优化。基于规则的研究一般是通过重写查询计划,去除查询的嵌套性来降低复杂度和子查询的执行次数来提高性能,基于代价的研究则是根据统计信息来进行查询计划的代价计算,选择代价最小的查询方案。

CBase 是西北工业大学联合交通银行、华东师范大学研发的分布式关系数据库,基于阿里巴巴的 OceanBase0.4.2,目前已经在交通银行上线应用[2-4]。CBase 目前仅仅支持部分相关子查询,且性能不佳。本文在对现有的查询优化策略进行分析研究的基础上,结合分布式数据库的特点,以查询重写为主,设计了相关子查询的优化策略,在 CBase 上实现了对于谓词 IN 和 EXISTS 相关子查询的上拉、无用分支切除和聚集函数消除的优化。

收稿日期:2020-11-25 **基金项目:**国家自然科学基金面上项目(61672432)与国家自然科学基金重点项目 61732014)资助

作者简介:张晨煜(1997—),西北工业大学硕士研究生,主要从事分布式数据库研究。

通信作者:刘文洁(1976—),女,西北工业大学副教授,主要从事云计算、大数据处理、海量分布式数据库研究。

e-mail:liuwenjie@ nwpu.edu.cn

1 相关研究

1.1 基于规则的研究

对于子查询,按照关键字的不同,可以划分为 3 类:①IN 为标志的集合成员查询;②ANY/SOME/ALL 关键词引导的比较查询;③以 EXISTS 为关键词的空判断查询。

对于这三类子查询,均可以等价为 EXISTS 语义的查询。在 MySQL 的研究中,将以 IN 为关键词的子查询,转换成了同义的 EXISTS 语句。该转换策略主要思路是将 IN 查询相关的列转换为 EXISTS 子句的等值过滤条件,遵循如下转换规则:

对于原有的 IN 查询:

SELECT C1 FROM T1 WHERE C2 IN (SELECT A1 FROM T2 WHERE T1.C3=T2.C3);

可以将其转换成等价的 EXISTS 语义的语句:

SELECT C1 FROM T1 WHERE EXISTS (SELECT 1 FROM T2 WHERE T1.C3 = T2.C2 AND T1.C2 = T2.A1);

不同于其他子查询需要扫描全表,EXISTS 子查询只需要取一次查询结果进行空判断。相较于其他的子查询,EXISTS 子查询是一种最优的子查询,因此在 MySQL 中,对于 IN 子查询的优化,采取了和 EXISTS 谓词相似的策略:

1) 查询展开,将子查询上拉;

2) 消除冗余子树;

3) 在相关列上构建索引。

而在 Oracle 的研究当中,Oracle 对子查询的优化处理也采用了反嵌套的思想,主要包括了 2 种类型的反嵌套,一种是生成派生表,另一种是合并子查询。接下来,将详细地讨论以上几种优化策略的思想。

1.1.1 子查询上拉

一个相关的 IN/EXISTS 子查询在语义上等价于一个半连接查询语句,将父查询块与子查询块的表格在父查询中做半连接操作,以子查询块中与父查询块相关的过滤条件作为连接条件,如果是 IN 子查询则 IN 引导的列也将作为连接条件。根据这一理论,Bellamkonda 等[5]提出了将相关子查询上拉展开为同语义的半连接查询的优化策略,通过对子查询的执行计划进行上拉,可以将子查询的嵌套层数减少一层,减少了相关子查询执行时需要多次运算子查询的时间,提高了查询性能。

但是这种优化策略对于待处理的子查询语句有着一定的限制,对于子查询中含有聚集函数、with 子句、子查询的 FROM 项为空或 EXISTS 子查询的 WHERE 项为空时,将无法采用这种优化策略。

除了可以上拉为半连接语句之外,还可以将子查询上拉为内连接查询。根据关系代数公式,假设有 2 个关系 $\boldsymbol{R}$ 与 $\boldsymbol{S}$,在属性 a 和 b 上进行半连接,用关系代数表示为 $\boldsymbol{R} \ltimes_{a=b} \boldsymbol{S}$,通过关系代数运算可以发现:$\boldsymbol{R} \ltimes_{a=b} \boldsymbol{S} = \Pi_R(\boldsymbol{R} \bowtie_{a=b} \boldsymbol{S}) = \boldsymbol{R} \bowtie_{a=b}(\Pi_b(\boldsymbol{S}))$。关系 $\boldsymbol{R}$ 和关系 $\boldsymbol{S}$ 在属性 a 和 b 上进行半连接的语义与 $\boldsymbol{R}$ 和 $\boldsymbol{S}$ 在属性 a 和 b 上进行内连接后对 $\boldsymbol{R}$ 的属性进行投影的语义是等价的。根据这条理论,由在文献[5-6]中指出可以将子查询上拉为内连接查询的优化策略,该优化策略同样可以减少子查询嵌套的层数。在上拉为内连接的优化策略中,将相关子查询分为了两类,一类为不含有聚集函数可以直接进行上拉的子查询,而另一类则是含有聚集函数的子查询。在该优化策略中,对于含有聚集函数的相关子查询,文献[5-6]中指出通过临时表的机制消除聚集函数,成为不含聚集函数的相关子查询后再进行上拉优化。

将子查询上拉为内连接相较于转换为半连接的策略,只解决了聚焦函数造成的优化限制,对于其余限制条件仍不能解决。

以上 2 种优化的主要思想基本上是一致的,都是对于待优化的相关子查询的查询计划中的父子查询块的 FROM 项进行连接,作为父查询的 FROM,再将子查询块 WHERE 中的过滤条件中与父表相关的属性作为连接属性上拉,并用 AND 连接,其余的子查询过滤条件则依次填充进父查询块的 WHERE 中,对于 IN 相关子查询,需要将 IN 相关的属性改为"="关系填充进连接条件,额外的,如果要转换为内连接查询,可能会有重复的结果出现,需要进行投影运算去重。在效果上,2 种方法也是基本一致的,都是将子查询的嵌套层数减去了内部的一层,减少了子查询块多次执行的时间开销。

1.1.2 冗余子树切除

对于相关子查询来说,由于 IN 相关子查询可以通过转换变为语义等价的 EXISTS 相关子查询,因此,对于子查询中的一些限制子句的处理,可以按照 EXISTS 的语义进行考虑。EXISTS 的语义对于子查询的结果只判断是否为空,并不关心结果中的顺序

等具体内容，所以子查询中的排序、去重等都是对于查询结果毫无贡献的无用操作，去除掉这些子句可以减少查询计划的内容，切除子树的规则如下：

例：原句为：

SELECT C1 FROM T1 WHERE C2 IN (SELECT DISTINCT(A1) FROM T2 WHERE T1.C3 = T2.C3 ORDER BY (A2) GROUP BY(A3))；

优化后为：

SELECT C1 FROM T1 WHERE C2 IN (SELECT A1 FROM T2 WHERE T1.C3=T2.C3)；

对于子句中的 DINSTINCT、GROUP BY、ORDER BY 谓词都可以切除以减少查询计划的路径长度，缩短时间开销。

1.1.3　相关列构建索引

对于相关子查询的优化，除了可以通过规则优化减少子查询的执行次数外，Shioi 等[7]提出了通过在 WHERE 项的属性上建立索引的方式来提升子查询执行的性能。这种优化策略，在子表数据量很大的情况下，可以大大缩短对子表访问所造成的时间开销，缩短查询时间。

1.1.4　子查询合并

子查询合并技术[5]指的是对于一个查询语句中，由 AND 或者 OR 谓词连接的多个子查询，可以在一定条件下将其合并为一个子查询，从而减少对表的访问次数。

如果一个子查询的结果集包含了另一个查询块的结果集，将其称为容器查询块，另一个称为包含查询块。对于同类型的 2 个子查询，如果满足包含关系，那么保留容器查询块，删去包含查询块。

当子查询块之间不满足包含与被包含的关系时，可以将具有等价语义的语义合并，并将其余下等非的条件合取或析取，使之成为一个子查询。

如果 2 个子查询块存在包含与被包含的关系，但是 2 个查询块类型不同。假定 EXISTS 是容器子查询块，需要在合并后引入一个 HAVING 子句，将满足 NOT EXISTS 条件的查询结果返回值设为 0，模拟其筛选过程。

1.1.5　聚集函数消除

对于聚集函数的处理策略是比较复杂的，在这里只讨论了一些简单的处理策略。在 EXISTS 中对于聚集函数的一些处理是简单的。由于 EXISTS 子查询的返回结果只是是否为空，而对于聚集函数的运算来说，无论是否有满足过滤条件的行，都会返回一个运算结果，即使是没有满足子查询过滤条件的元组，也会返回一个 0 或者 NULL 的结果，而非空结果，这就说明聚集函数在 EXISTS 的子句中是返回一个恒为 TRUE 的值的，即子查询恒成立，所以可以将含有聚集函数的子查询删去，减少子查询的执行。

例：

SELECT * FROM T1 WHERE T1.C2 = 1 AND EXISTS(SELECT COUNT (*) FROM T2 WHERE T1.C1=T2.C1)；

优化后的结果为：

SELECT * FROM T1 WHERE T1.C2=1；

这种优化策略删去了对结果毫无影响的含有聚集函数的子查询，消去了子查询执行的时间，在面对可优化的情况下，性能比较显著，但是可优化的类型比较局限。

1.2　基于代价的研究

基于代价的优化需要引入统计信息、直方图等的功能[8-10]。基于代价的优化需要统计信息提供的关于表的相关信息，预先的估算查询可用的若干计划的可能开销，从中选择出代价最小的执行方案执行。

由于本文暂时只考虑了基于规则的优化设计，所以在此对于目前基于代价的研究只进行简单的讨论。

在商业数据库 DBMS-X[11]中提出了一种名为位向量过滤器的优化措施。这种优化主要用于哈希连接，在哈希连接各个表的连接操作符处创建位向量过滤器，并将其下推到计划中可能与该操作符涉及到的表相关的表或者其他位向量过滤器上。位向量过滤器的应用并非对已经过代价计算选择出的最优计划直接添加维向量过滤器，而是在代价计算中直接考虑了维向量过滤器的影响。因为相关的实验已表明，在考虑了维向量过滤器后生成的查询计划的代价，远小于对最优计划添加维向量过滤器后的代价，这也就说明，对于位向量过滤器的考虑，需要在基于代价的优化过程中进行，而位向量过滤器的引入，也可能破坏了最优子计划的原则，破坏了现有的动态规划框架，因为考虑位向量过滤器的最小代价计划，在删去位向量过滤器后，代价可能是高于不考虑位向量过滤器的情况下生成的执行计划的。

在 Postgre SQL[12]中，提出了一种子查询计划重用的方法，来提高动态规划得出最优执行计划的代价。其主要思想是优化了得出最优执行计划的计算

开销和空间开销。这种策略允许了子查询在搜索空间中共享相似子查询的查询计划。这种优化策略的主要思想在于将每个查询抽象成一个与之对应的图,并且生成他的子图覆盖集,对于后来的查询,在子图覆盖集中发现与之同构的子图,共享对应的查询计划。而生成子图覆盖集是一个内存密集的运算,在该研究中,对于覆盖集的生成也进行了优化,减少了内存开销。

1.3 分布式系统中的优化

分布式数据库系统与集中式数据库系统之间查询代价的差异,主要体现在查询代价的构成上。传统的集中式数据库,查询代价由 CPU 执行时间与 I/O构成。而分布式数据库由于各个节点分布在网络当中,除了基础的 CPU 时间和 I/O 外,还有各个节点之间的网络通信耗时。所以分布式数据库在优化策略的设计上,除了结合自身分布式特点的策略外,还借鉴了传统的数据库。

分布式数据库的查询优化主要考虑 2 个方面:减少查询耗时和通过分布式特点并行处理查询。

查询耗时的优化主要是通过对查询路径的优化减少磁盘块的传递次数和对扫描次数进行优化减少对磁盘块的扫描次数[13]。这一目标的优化方案通常借鉴了传统数据库的方法,对逻辑计划和物理计划进行优化。

而并行处理则是结合了分布式的特点,提出了一种以操作符为中心的数据流模型[14],将生成的查询计划按照操作符进行拆分,每个操作符都对应一个引擎,分发到各个服务器进行计算,最后将结果合并。通过较低的计算代价和数据冗余存储换取高昂的网络开销,把一个查询分到数据节点上进行并发查询,最后在主节点融合数据结果,使得总体响应时间很短。同时,考虑到查询批处理[10]的情况,对于每个引擎设置了一个可以接受的延迟时间,将若干查询中对于同一个服务器的数据请求整合起来进行访问,减少了多次访问数据造成的网络开销。

2 算法实现

对于分布式数据库优化的 2 个目标,本文主要关注于第一个目标,减少查询时间消耗,参考传统数据库对于查询优化的一些思想,进行基于规则的优化。

本文的思想是通过聚集函数消除和切除子查询中的无用子树,优化查询路径,减少磁盘块的传递次数,减少各服务站点之间的网络开销。通过将相关子查询上拉为连接查询,减少对子查询表的扫描次数和反复将子表数据传输的网络开销。

2.1 实现背景介绍

本文的研究环境基于分布式数据库 CBASE,其架构中分为 4 类服务器,分别是:RootServer、ChunkServer、MergeServer 和 UpdateServer 。数据包括基准数据和增量数据,基准数据存储在 ChunkServer 中,增量数据存储在 UpdateServer 中,对于查询的处理,将从 MergeServer 收到查询请求后,根据 RootServer 上的信息,访问相关的 ChunkServer 和 UpdateServer 的数据,在 MergeServer 上合并后返回给用户。对于查询表的访问,除了含有主键的查询会采用 GET 方法直接访问表中的对应元组外,其他的过滤条件都将采用 SCAN 的方式逐行扫描查询表中的元组。这就导致了被访问的表的元组数越多,时间开销越大。在面对嵌套查询的时候,多次访问子表将会造成极大的时间开销。

在目前的 CBASE 环境中,只支持了 EXISTS 的子查询功能和 IN 的非相关子查询功能,本文参考 MySQL 中的转换规则实现 IN 相关子查询并对 IN 和 EXISTS 相关子查询进行优化。

2.2 现有 IN 子查询执行策略

本文将详细讨论并实现 IN 相关子查询的方法,并介绍在未优化的情况下 IN 相关子查询与 EXISTS 相关子查询的执行方法。

对于 IN 相关子查询,考虑可以通过转换现有的执行计划,使其成为等价的 EXISTS 相关子查询,转换的普遍规则如下:

原查询:

SELECT expr FROM TABLE_1 WHERE outexpr1, outexpr2, …, outexprn IN(SELECT innerexpr1, innnerexpr2, …, innerexprn FROM TABLE_2 WHERE where_expr);

转换后查询:

SELECT expr FROM TABLE_1 WHERE EXISTS (SELECT 1 FROM TABLE_2 WHERE where_expr AND outexpr1 = innerexpr1 AND outexpr2 = innerexpr2 …AND outexprn = innerexprn);

由于 EXISTS 子查询的返回结果只是空与非空,对具体内容并不关心,所以子查询的目标并不需要指定,为 1 即可, 保留原有的子查询的过滤条件,

将原有查询中 IN 相关的属性一一对应改为“ = ”作为子查询的新过滤条件。

这是最为普遍的转换方法,额外的,需要讨论被转换的查询中,IN 相关的属性允许为 NULL 值的情况。

当 innerexpr 可能为 NULL 值时,在添加过滤条件 outexpr = innerexpr 时,要析取一个判断 innerexpr IS NULL,将析取后的结果作为过滤条件添加进子查询的 WHERE 中。假设 innerexpr1 可能为 NULL,

例:

SELECT expr FROM TABLE_1 WHERE EXISTS (SELECT 1 FROM TABLE_2 WHERE where_expr AND(outexpr1 = innerexpr1 OR innerexpr1 IS NULL) AND outexpr2 = innerexpr2… AND outexprn = innerexprn);

在完成上述转换后,在物理计划的执行上对于相关子查询是类似的,物理计划会首先执行主查询块的查询,取出主查询中的每一个元组,对于多个过滤条件的,如果是用 OR 连接的过滤条件,则只需判断其余过滤条件,如果其余过滤条件为假,再将该元组传递给子查询执行,如果为真,则可以不用执行子查询。对于由 AND 连接的过滤条件,需要判断是否满足其他过滤条件,只有满足其他条件的元组信息会被传递给子查询执行。

对于传递给子查询的元组信息,物理计划会将子查询中主查询相关的属性改写成元组中具体的值,然后根据重写后的物理计划进行执行计算子查询的结果,如果子查询执行后存在满足条件的结果,将返回给主查询一个 TRUE 值,否则返回 FALSE 值。对于可以使子查询结果为 TRUE 的元组,会将该元组的信息输出,然后取出下一行元组,如此进行直到将主查询中的所有元组扫描完成。

目前的执行流程如图 1 所示。

从流程图中可以直观的看出,现有的执行策略对于每一个子查询都有着嵌套的 2 个循环过程,对于外层父查询的每一个元组都需要经过一次完整的子查询执行流程,这在外层查询的元组增加时大量的增加时间开销,同时,如果存在多层嵌套的子查询,那么时间开销更是会指数级增加,反复对子查询的执行,也将会导致数据频繁的进行通信,在分布式的环境下也会产生大量的通信时间开销,接下来,将讨论对子查询优化的具体策略与实现方法。

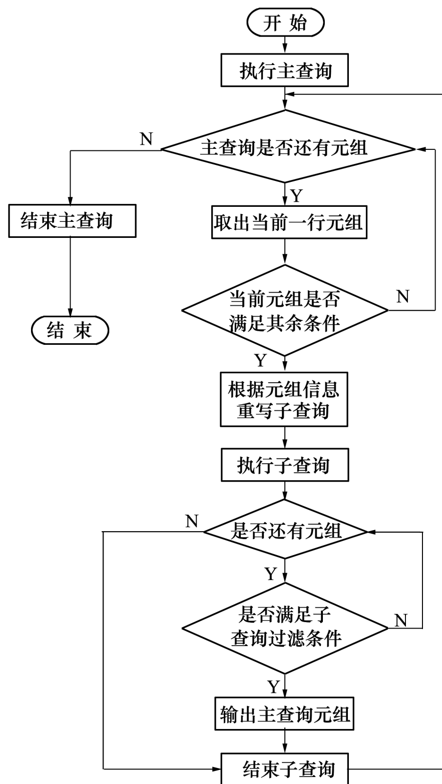

图 1　现有子查询执行流程图

2.3　优化策略

对分布式数据库不同的语句环境,本文选择了 3 种优化方案:子查询冗余子树消除、聚集函数消除和子查询上拉方案,针对不同的情况进行优化。

2.3.1　子查询冗余子树切除

对于 IN 和 EXISTS 的相关子查询来说,IN 子查询等价于在原有子查询的过滤条件中添加了一个 IN 内外表达式的等值过滤,在执行时只是判断是否对于过滤条件有非空结果产生,因此,子查询的结果集是否排序、是否按某一属性进行分组、选择对象是否有唯一性限制并不会对查询的结果产生影响。然而,这些子句会在物理计划的生成与执行时产生多余的执行路径与操作符,进而导致执行时间开销增加,所以对于子查询中的去重限制,无其他子句的 group by 子句和 order by 子句进行切除,减少子查询的操作。切除示例:

SELECT C1 FROM T1 WHERE C2 IN(SELECT DISTINCT(A1) FROM T2 WHERE T1.C3 = T2.C3 ORDER BY (A2) GROUP BY(A3));

优化后为:

SELECT C1 FROM T1 WHERE C2 IN (SELECT A1 FROM T2 WHERE T1.C3 = T2.C3);

2.3.2 聚集函数消除

对于涉及到聚集函数的大部分情况来说,要消除聚焦函数是比较复杂的,本文只实现了对于EXISTS相关子查询的聚集函数消除。聚集函数的计算结果对于即使是表中不存在满足过滤条件的元组,也会返回一个有着具体值的结果,比如count()会返回0值,sum()会返回NULL值,然而无论返回什么样的值,结果总是非空的,对于EXISTS的子查询来说,恒为非空的子查询将总会允许主查询的元组被输出,所以带有聚集函数的子查询并没有对整个查询产生影响,因此可以在构造计划时,删去含有聚集函数的子查询。

例:SELECT * FROM T1 WHERE T1.C2=1 AND EXISTS(SELECT COUNT(*)FROM T2 WHERE T1.C1=T2.C1);

优化后:

SELECT * FROM T1 WHERE T1.C2=1;

2.3.3 子查询上拉

由于在语义上,IN相关子查询和EXISTS相关子查询是等价的,又都等价于半连接语义,所以可以考虑将子查询中的表与条件展开上拉到主查询中,这样可以使得嵌套的子查询变成了一层的连接查询,减少了查询执行的次数。

由于目前的CBASE中暂不支持半连接语义的查询,由1.1.1节的公式可知,2个关系 ***R*** 与 ***S*** 在某一属性上的半连接,等价于对 ***R*** 与 ***S*** 在该属性上进行内连接后对关系 ***R*** 的属性进行投影。所以在这里采用内连接的策略,将IN相关子查询的内容,上拉合并到主查询中。

将IN子查询的表作为连接表合并进主查询的FROM中,用INNER JOIN连接,将子查询中与主查询相关的过滤条件作为连接条件,其余过滤条件上拉合并至主查询的过滤条件中。对于IN相关的内外表达式,依次构造成等值过滤条件,添加进连接条件中,对于主查询的目标,需要进行额外的去重操作。

但是这种优化策略仅支持简单的相关子查询,要求子查询中不含有聚集函数,不含有with子句,且子查询的FROM和WHERE内容不能为空,否则无法构造连接查询。具体如下:

例:

SELECT C1 FROM T1 WHERE C2 IN(SELECT A1 FROM T2 WHERE T1.C3=T2.C3 AND T2.A2=3);

优化后:

SELECT DISTINCT (C1) FROM T1 INNER JOIN T2 ON T1.C2=T2.A1 AND T1.C3=T2.C3 WHERE T2.A2=3;

子查询上拉为连接查询的优化方法虽然对于待优化的语句类型有着较为严格的限制,局限于简单的子查询语句,但是在日常的使用中,简单的子查询语句占了很大的比例,而且相关子查询在面对查询数据的增加时,耗时也会随之剧增,因此,即使子查询上拉只针对于简单子查询,也有着很大的应用环境。

2.4 优化算法设计

对于优化的进行,需要在解析语法树的过程中进行一些准备工作,将查询语句中的IN/EXISTS查询块的外部表达式和子查询的查询对象存储起来预备进行上拉优化。优化的预处理工作与优化器的调用流程如图2所示。

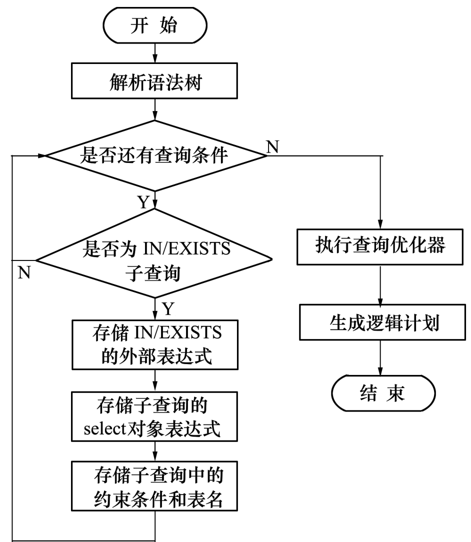

图2 优化预处理及优化器流程图

本文将综合2.3节中讲述的几种策略进行优化,对于每一个进入优化器的查询,将依次处理每一个子查询块,先对输入的子查询可切除冗余子树进行切除,之后,对含有聚集函数的EXISTS子查询进行消除。经过前2步处理后的子查询,如果存在满足可优化的简单子查询,进行上拉展开,子查询中还含有子查询,则递归地调用优化器,从最底层的子查询依次向上优化展开,减少查询层数。伪代码如下:

Optimize(query){

```
For each subquery_i{
    If(subquery_i 存在冗余子树){
        切除冗余子树;
    }
    If ( subquery_ i 存在聚集函数且是EXISTS){
        消除 subquery_i;
    }else(subquery_i 是简单相关子查询){
        If(是叶子子查询){
            subquery_i FROM 项上拉;
            subquery_i WHERE 与父查询相关项上拉为连接条件;
            subquery_iWHERE 剩余项上拉;
            subquery_i in 相关项重构为等值语句上拉为连接条件;
        }else{
            Optimize(subquery_i);
        }
    }
  }
}
```

3　实验设计与结果

3.1　实验环境

实验选择在 Linux 服务器上部署的 CBASE 分布式数据库上进行,Linux 服务器环境参数如表 1 所示。

表 1　测试机器配置

操作系统	Red Hat Enterprise Linux Server release 6.2
内核版本号	Linux version 2.6.32-220.el6.x86_64
内存	16G
CBase 版本	CBase Version:0.4.2.21 单集群
CPU	4 核
硬盘	1T
TPC-H 数据量	1 万,10 万

性能测试采用了 TPC-H 基准的 3 张表:nation、supplier 和 customer 并生成测试数据。指定 TPC-H 生成总量为 1G 的随机数据,其中,nation 表数据量为 25 行,supplier 表数据量为 1 万行,customer 表数据为 10 万行以上。以 nation 表作为外表,其他 2 张表作为构成子查询的内表,分别用来进行不同数据量的测试。实验目的是验证目前选择的 3 种策略,在单独作用的情况下,是否能减少查询的执行时间。针对 3 种情况设计了不同的 SQL 语句来进行验证优化前后的性能。对于聚集函数消除,由于这种优化方案针对的情况与其他 2 种方案并不重合,所以仅设计单独的情况来验证性能。对于冗余子树切除和子查询上拉,除了分别验证各自的性能外,还设计了 2 种情况的 SQL 语句,测试综合性能。在实验结果展示中,将分别表示出对不同类型语句优化前后的时间开销,以及对冗余子树切除后仍能进行上拉优化的语句再次优化的效果,称之为二次优化。

为了使性能测试的结果更接近实际用途,测试表的数据量外表 25 行数据,内表 10 000 数据和外表 25 行数据,内表 100 000 数据测试集。

3.2　实验结果

1) 25×10 000 数据

外表的内容为:

nation(N_NATIONKEY, N_NAME, N_REGIONKEY, N_COMMENT);

内表的内容为:

supplier(S_SUPPKEY,S_NAME,S_ADDRESS,S_NATIONKEY,_PHONE,S_ACCTBAL,S_COMMENT);

设计的测试语句为:

针对 distinct 子句切除:

select * from nation where exists (select distinct(s_name) from supplier where supplier. s_ nationkey = nation.n_nationkey);

针对 group by 子句切除:

select * from nation where n_nationkey in(select s_nationkey from supplier where nation. n_ regionkey = 1 group by (s_suppkey));

针对 order by 切除:

select * from nation where n_nationkey in(select s_nationkey from supplier where nation. n_ regionkey = 1 order by (s_suppkey));

聚集函数消除:

select * from nation where exists(select count(*) from supplier where supplier.s_nationkey=nation.n_nationkey);

上拉优化:

select * from nation where n_nationkey in (select s_nationkey from supplier where nation.n_regionkey=1);

对于这几种优化的实际执行结果如图 3 所示。

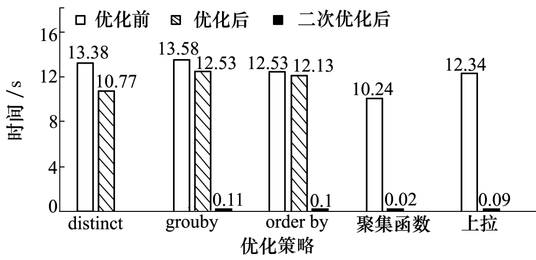

图 3　25×1 万数据优化性能显示

可以从实验的结果中直观地看出,聚集函数由于直接切除了整个子查询,在无其他子查询时,相当于对主查询的表进行一个简单的扫描,所以优化性能非常显著,但是这种优化方案的应用环境非常受限,适用不广。冗余子树的切除虽然对于性能有一定的提升,但是可以看出提升并不显著。切除 dinstinct 路径的优化有 19%,切除 groupby 和 orderby 的性能提升比较接近,都是 5%左右。可以看出,在冗余子树切除之间相比,distinct 切除后的性能优化更为明显,这说明在进行 distinct 运算时需要耗费较多的计算时间,所以其优化性能相比其他的运算略微明显。而由于设计的实验语句,均为相关子查询,即使通过冗余子树切除减少了不必要的查询路径,优化后的语句仍是一个相关子查询语句,冗余子树进行计算的 CPU 开销和获取相关数据的磁盘开销,相比于相关子查询对子表进行反复扫描的磁盘 I/O 和网络开销,占了总耗时中很少的一部分,所以在冗余子树切除后,实验语句的耗时仍然是较长的。子查询展开转换成连接的优化方案,可以看出,无论是单独的子查询上拉的情况,还是先进行冗余子树切除后再进行上拉的环境,都有着不错的性能,这是由于相关子查询对于外表中每一个符合条件的元组,都要进行一次子表扫描,这样的耗时是非常巨大的,将子查询上拉后,只需要进行一次扫描做连接运算,这大大减少了对磁盘的扫描和网络通信。

2) 25×100 000 数据:

外表内容:

nation(N_NATIONKEY, N_NAME, N_REGIONKEY, N_COMMENT);

内表内容:

customer(C_CUSTKEY, C_NAME, C_ADDRESS, C_NATIONKEY, C_PHONE, C_ACCTBAL, C_MKTSEGMENT, C_COMMENT);

针对 distinct 切除:

select * from nation where exists (select distinct(c_custkey) from customer where customer.c_nationkey = nation.n_nationkey);

针对 group by 切除:

select * from nation where n_nationkey in (select c_nationkey from customer where nation.n_regionkey = 1 group by (c_name));

针对 order by 切除:

select * from nation where n_nationkey in (select c_nationkey from customer where nation.n_regionkey = 1 order by (c_name));

聚集函数消除:

select * from nation where exists (select count(*) from customer where customer.c_nationkey = nation.n_nationkey and nation.n_regionkey = 1);

子查询上拉:

select * from nation where n_nationkey in (select c_nationkey from customer where nation.n_regionkey = 1);

实验结果如下所示:

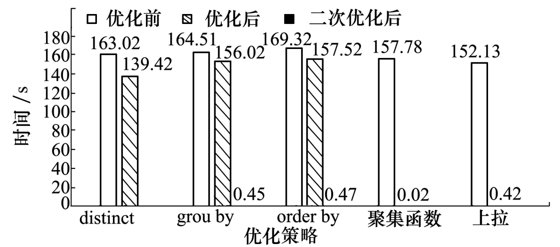

图 4　25×10 万数据优化性能显示

同样的,在数据量增多的情况下可以发现同样的结论,此外,也可以看出,对于冗余子树切除的优化,随着数据量的增多,其性能的提升略微有所下降,这是由于数据量的增多使得每次扫描子表的开销随之增加,这就导致了相关子查询多次扫描子表内容产生的开销占据了更大的比重,因此子查询上拉的优化策略随着数据量的增多显得性能更为显著。

可以发现,单独的冗余子树切除对于查询性能的提升只有不到 20%,而且在子查询存在有需要上拉优化的情况下,查询内外表的数据量越多,子查询在切除子树后越复杂,冗余子树切除对查询性能的优化效果越不明显。聚集函数消除仅针对特定情况下的查询有作用,但是对于可进行优化的查询效果提升非常显著,优化后的性能只与主查询的复杂程

度相关。上拉优化是进行子查询优化的主要方案,目前大部分常用的查询语句都属于上拉优化可优化的范围,也可以从实验中看出,上拉优化对于相关子查询的时间开销优化作用显著,可以达到90%以上。本文优化方案在当前的CBASE环境下,面对特殊的情况和大部分普遍情况都是可以进行效果显著的优化的。

4 结 论

本文研究了目前现有的数据库优化策略,结合分布式数据库CBASE的特点,选择了子查询上拉为内连接,聚集函数消除,冗余子树切除的方案,针对目前CBASE中IN相关子查询执行性能差、时间开销大的问题,提出了优化方案。本文的优化方案从理论上可以减少IN相关子查询执行时不必要的物理计划操作,减少了查询的嵌套层次,转为层数较少的连接查询。

实验结果表明:3种方案在面对特殊的聚集函数消除的情况时效果最为显著;在面对普遍常见的简单子查询时,上拉优化也可以有着显著的优化效果;冗余子树切除的优化策略在被优化子查询数据量大的情况时效果愈发平庸。

本文的研究中只针对了IN相关子查询中的简单子查询和EXISTS查询中的聚集函数的优化,且只实现了基于规则的优化。对于其他更为复杂的查询语句以及基于代价选择的优化,本文暂时还没有提出解决方案,在后续的工作中,将继续深入研究视图消除,子查询合并等优化方案 ,并对基于代价选择的优化提出解决方法,此外,将采用MPP架构,将查询计划按照操作符进行拆分,进行并行处理的优化。

参考文献:

[1] 萨师煊, 王珊. 数据库系统概论[M]. 北京:高等教育出版社
SA Shixuan, WANG Shan. Database system concepts[M]. Beijing: Higher Education Press (in Chinese)

[2] 刘文洁, 李戬勃, 李战怀. 一种面向金融应用的海量分布式关系数据库[J]. 华中科技大学学报, 2019,47(2): 121-126
LIU Wenjie, LI Jianbo, LI Zhanhuai. A massive distributed relational database for financial application[J]. Journal of Huazhong University, 2019, 47(2): 121-126 (in Chinese)

[3] 高锦涛, 刘文洁, 李战怀. 一种面向分布式读写分离系统的数据同步策略[J]. 西北工业大学学报, 2020, 38(1): 209-215
GAO Jintao, LIU Wenjie, LI Zhanhuai. A strategy of data synchronization in distributed system with read separating from write [J]. Journal of Northwestern Polytechnical University, 2020, 38(1): 209-215 (in Chinese)

[4] 高锦涛, 李战怀, 杜洪涛, 等. 分布式数据库下基于剪枝的并行合并连接策略[J]. 软件学报, 2019, 30(11): 3364-3381
GAO Jintao, LI Zhanhuai, DU Hongtao, et al. Strategy of parallel merge join based on prune in distributed database[J]. Journal of Software, 2019, 30(11): 3364-3381 (in Chinese)

[5] BELLAMKONDA Srikanth, AHMED Rafi, ANDREW Witkowski, et al. Enhanced subquery optimizations in oracle[J]. Proc VLDB Endow, 2009, 2(2): 1366-1377

[6] 毛思雨, 张利军, 张小芳, 等. 面向分布式数据库的相关子查询优化策略[J]. 华东师范大学学报,2016(5): 57-66
MAO Siyu, ZHANG Lijun, ZHANG Xiaofang, et al. Optimization strategies of correlated subquery for distributed database[J]. Journal of East China Normal University, 2016(5): 57-66 (in Chinese)

[7] SHIOI Takamitsu, HATANO Kenji. Query processing optimization using disk-based row-store and column-store[J]. iiWAS, 2015, 69: 1-9

[8] MOHAMMED H A, MURAT K. To review and compare evolutionary algorithms in optimization of distributed database query[C] //International Symposium on Digital Forensics and Security, 2020: 1-5

[9] ESLAMI Mehrad, MAHMOODIAN Vahid, DAYARIAN Iman, et al. Query batching optimization in database systems[J]. Computers & Operations Research,2020,121: 1-17

[10] JI Xuechun, ZHAO Maoxian, ZHAI Mingyu, et al. Query execution optimization in spark SQL[J]. Sci Program, 2020, 2020: 6364752

[11] DING Bailu, CHAUDHURI Surajit, NARASAYYA Vivek R. Bitvector-aware query optimization for decision support queries[C]

//SIGMOD Conference, 2020: 2011-2026
[12] VAMSIKRISHNA Meduri Venkata, TAN Kian-Lee. Subquery plan reuse based query optimization[C]//International Conference on Management of Data, 2011: 35-46
[13] 李海翔. 数据库查询优化器的艺术[M]. 北京:机械工业出版社, 2014
LI Haixiang. The art of database query optimizer principle and SQL performance optimization[M]. Beijing: China Machine Press, 2014 (in Chinese)
[14] CHEN G, WU Y, LIU J, et al. Optimization of sub-query processing in distributed data integration systems[J]. Journal of Network and Computer Applications, 2011, 34(4): 1035-1042

Optimization of correlate subquery based on distributed database

ZHANG Chenyu[1], LIU Wenjie[1], PANG Tianze[2], YUE Yantao[2]
(1.School of Computer Science, Northwestern Polytechnical University, Xi'an 710072, China;
2.Bank of Communications, Shanghai 200120, China)

Abstract: Subquery is widely used in database. It can be divided into related subquery and non-related subquery according to whether it is dependent on the table of the parent query. For related subqueries, it is necessary to take a tuple from the parent query before executing the subquery, that is, the content of the subquery needs to be repeatedly operated. Disk access costs of this strategy is very big, in the distributed database, because of data communication overhead, in the parent query yuan set is too low efficiency, therefore, for the class sub queries, on the basis of the optimization of the existing query strategy, combining with the characteristics of distributed database, put forward by the subquery on to join queries, eliminate redundant clauses in the subquery, eliminate accumulation function method based on distributed database query optimization strategy, and the effectiveness of the present optimization strategy is verified by experiment.

Keywords: Distributed database; Correlate subquery optimization; rule-based optimization

引用格式:张晨煜, 刘文洁, 庞天泽, 等. 基于分布式数据库的相关子查询优化[J]. 西北工业大学学报,2021,39(4): 909-918
ZHANG Chenyu, LIU Wenjie, PANG Tianze, et al. Optimization of correlate subquery based on distributed database[J]. Journal of Northwestern Polytechnical University, 2021, 39(4): 909-918 (in Chinese)

© 2021 Journal of Northwestern Polytechnical University.

This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.