

一种面向数据仓库周期性查询的增量优化方法*

康炎丽^{1,2}, 李丰¹, 王蕾^{1,2}



¹(计算机体系结构国家重点实验室(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 100190)

通讯作者: 李丰, E-mail: lifeng2005@ict.ac.cn

摘要: 大数据蕴含着巨大的价值. 分析类查询是获取数据价值的一种重要手段. 为及时把握分析结果的变化, 查询需要周期性地重复. 为此, 将不可避免地引入对旧数据的重复分析. 目前, 以重用历史数据的中间结果、优化冗余计算为核心思路的增量分析技术, 存在用户透明性不佳、对历史结果存储位置的选择不够智能化等问题, 对周期性增量查询的优化效果有限. 从兼顾用户透明性和优化收益的角度出发, 设计了一种以语义规则为指导的增量优化方法. 该方法扩展了增量描述语法, 以查询操作符的操作语义和输出语义指导对历史数据存储、合并位置的选择, 再根据代价模型和物理查询任务的划分位置对选择结果进行调整, 生成优化后可以在分布式计算框架(如 MapReduce)周期性调度执行的物理查询任务. 以 Apache Hive 为基础, 实现了上述方法的原型 HiveInc. 实验结果表明: 对于扩展了增量语法描述的 TPC-H 测试集, HiveInc 相对于优化前可以获得平均 2.93 倍、最高 5.78 倍的加速; 与经典的优化技术 IncMR、DryadInc 相比, 分别可以获得 1.69 倍和 1.61 倍的加速.

关键词: 数据仓库; 周期性查询; 增量优化; 中间结果重用

中图法分类号: TP311

中文引用格式: 康炎丽, 李丰, 王蕾. 一种面向数据仓库周期性查询的增量优化方法. 软件学报, 2017, 28(8): 2126-2147. <http://www.jos.org.cn/1000-9825/5107.htm>

英文引用格式: Kang YL, Li F, Wang L. Incremental optimization method for periodic query in data warehouse. Ruan Jian Xue Bao/Journal of Software, 2017, 28(8): 2126-2147 (in Chinese). <http://www.jos.org.cn/1000-9825/5107.htm>

Incremental Optimization Method for Periodic Query in Data Warehouse

KANG Yan-Li^{1,2}, LI Feng¹, WANG Lei^{1,2}

¹(State Key Laboratory of Computer Architecture (Institute of Computing Technology, The Chinese Academy of Sciences), Beijing 100190, China)

²(University of Chinese Academy of Sciences, Beijing 100190, China)

Abstract: Analytical query is an important way to get value from big data in data warehouse. With the growth of data, the same query needs to be executed periodically, which inevitably introduces redundant calculation on historical data. One type of incremental optimization technology reduces redundant calculation by reusing intermediate results of historical data. However it has following problems: 1) it isn't transparent for user; 2) choice of historical result storing/reusing position is not intelligent; and 3) optimization gains is limited. This article designs an incremental optimization method, which is guided by the semantic rules. This method focuses on both user transparency and optimization gains, and extends grammar to support incremental description. Historical result storing/reusing location is firstly chosen by operators' operational semantics and output semantics. Positions are then adjusted according to cost model

* 基金项目: 国家高技术研究发展计划(863)(2015AA011505); 国家自然科学基金(61303053, 61402445, 61402303, 61521092)

Foundation item: National High-Tech R&D Program of China (863) (2015AA011505); National Natural Science Foundation of China (61303053, 61402445, 61402303, 61521092)

收稿时间: 2016-03-31; 修改时间: 2016-05-12; 采用时间: 2016-05-28; jos 在线出版时间: 2016-10-11

CNKI 网络优先出版: 2016-10-12 16:26:54, <http://www.cnki.net/kcms/detail/11.2560.TP.20161012.1626.022.html>

and physical task's division positions. At last, optimized tasks-DAG is generated with the ability to run in a distributed computing framework (such as MapReduce) periodically. This paper implements a prototype, called HiveInc, based on Apache Hive. Experimental results on TPC-H show that, compared to non-optimization, HiveInc can obtain average 2.93 speed-up and highest 5.78 speed-up. Compared to classical optimization techniques, IncMR and DryadInc, speed-up of 1.69 and 1.61 can be obtained respectively.

Key words: data warehouse; periodic query; incremental optimize; middle result reusing

信息时代,各个数据中心、互联网公司都存储着大量数据(比如用户活动日志、网页库等).以 Facebook 为例,其每天新增数据量超过 500TB^[1].新收集到的数据会周期性地(比如每隔 1 小时、每隔 1 天等)添加到数据仓库中.这些持续增长的数据蕴藏了巨大的价值.比如,广告系统的用户活动日志中包含用户编号、广告编号、是否曝光、是否点击等信息,通过分析每条广告的曝光量、点击量、广告费等信息,可以使广告主了解其所投放的广告实时状态;又如,电影评分网站利用网络爬虫收集论坛中的影评,实时更新各个电影的评分,使得想要观影的用户及时获得各个电影的最新评价.

大数据分析查询系统 Hive^[2]是目前最常用的分布式数据仓库查询引擎之一,最适合处理大批量的分析类查询任务.Hive 提供简洁且功能强大的类 SQL 查询语法(HiveQL).用户使用 HiveQL 语法书写的查询,经过 Hive 解析、优化后,转化为一系列能够在不同分布式计算框架(比如 MapReduce^[3],Spark^[4],Tenzing^[5])上执行的分布式查询任务(如图 1 所示).图中的抽象语法树(AST)、查询块(QB)和操作符树(OpTree)为 Hive 逻辑查询阶段的 3 种中间表示形式.

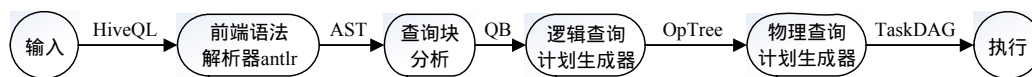


Fig.1 Architecture of Hive

图 1 Hive 原理图

由于数据仓库里的数据是周期性增长的,为了及时把握分析结果的变化,对数据分析查询也需要周期性地重复.本文将这种周期性重复地对数据仓库中不断增长的数据的分析场景称为周期性增量查询场景.在此过程中,势必会引入对旧数据的重复分析.以图 2 所示的查询语句为例,该查询取自标准测试集 TPC-H,用 HiveQL 语法书写,作用是分析巴西的电镀钢材在美洲市场中占的市场份额.该查询涉及的 8 张表中,以 lineitem 表(别名 l)的规模最大,是规模次大的 orders 表的 4.5 倍.查询语句中,加粗的注释为本文针对周期性增量查询场景扩展的专用描述语法(详见第 2.1 节).该语法描述的功能是指定从 1995 年 1 月 1 日起,每隔一季,重复执行一遍查询.假定每季度的数据量增长相当.如果每次重复该查询,都分析从 1995 年 1 月 1 日起到查询执行时间点之间的所有数据,则仅仅当年重复扫描的数据量就会达到 lineitem 表规模的 6 倍,由于重复分析历史数据而引入的时间开销为第 1 季度查询开销的 3.4 倍.

```

select o_year, sum(case when nation='BRAZIL' then volume else 0.0 end)/sum(volume)
from
(
  Select year(o_orderdate) as o_year, l_extendedprice*(1-l_discount) as volume, n2.n_name as nation
  from
    lineitem /*+INCREMENTAL(after 1995/1/1 INTERVAL 90/days)*/ l
    join part p on p.p_partkey=l.l_partkey and p.p_type='ECONOMY ANODIZED STEEL'
    join supplier s on s.s_suppkey=l.l_suppkey
    join nation n1 on s.s_nationkey=n1.n_nationkey
    join orders o on l.l_orderkey=o.o_orderkey and o.o_orderdate <= '1995-01-01'
    join customer c on c.c_custkey=o.o_custkey
    join nation n2 on c.c_nationkey=n2.n_nationkey
    join region r on n2.n_regionkey=r.r_regionkey and r.r_name='AMERICA'
) all_nation
group by o_year
order by o_year;
  
```

Fig.2 Example for Hive incremental query

图 2 Hive 增量查询示例

为避免由重复分析历史数据而引入的不必要的开销,需要为主流的大数据分析与查询系统,扩展针对周期性增量查询场景的支持.当前,针对周期性增量查询场景的处理支持,以重用历史数据的中间结果、优化冗余计算为核心设计思路,下文统称为增量优化技术.增量优化技术,按照实现层次,可以分为两大类:(1) 基于分布式查询系统所依托的分布式计算框架(也称为查询的执行引擎)设计的增量优化技术,代表性工作有 Percolator^[6], CBP^[7], IncMR^[8], Incoop^[9], DryadInc^[10]; (2) 针对有向无环图(DAG)形式的查询 workflow 设计的增量优化技术,代表性工作有 Nectar^[11], Nova^[12].这些增量技术虽然各具优势,但在用户透明性和优化收益方面还存在改进空间.部分技术(如 CBP, Nova 等)需要用户参与优化过程,比如,需要用户来选择历史数据中间结果存储点、设计增量处理逻辑,因此,不具备用户透明性.其他技术(如 IncMR, DryadInc 等)或者会存储多余的历史数据中间结果,为优化过程带来额外的开销,或者不能最大限度地重用历史结果,造成优化收益不理想.

本文针对上述问题,从兼顾用户透明性和优化收益的角度出发,设计了一种以查询操作符的语义特征和增量数据特点为指导的、查询执行引擎无关的周期性增量优化技术(下文简称语义规则指导的增量优化技术).该技术首先扩展了查询语法,供用户描述查询的重复周期和增量表.下文将该扩展语法称为增量描述语法.通过解析增量描述语法,本文在构建查询计划的过程中,将查询划分成增量无关部分和增量相关部分,并在保持查询语义不变的前提下,尽可能地扩大增量无关部分.然后,结合基于查询操作符的计算语义和输出语义设计的增量优化规则(第 3.1 节)、代价模型(第 4.3 节)以及物理查询任务的划分结果,在增量相关部分识别历史数据中间结果的存储位置、历史数据中间结果与新增量数据中间结果的合并位置.根据识别结果,在查询计划中自动插入用于完成存储、合并功能的操作符子树,生成优化后的增量查询计划.最后,将增量无关部分的查询计划和增量查询计划分别转化成可以在主流分布式计算框架上执行的查询任务,前者仅执行 1 次,后者按照增量描述语法所描述的周期调度执行.

本文在主流的分布式查询系统 Hive 上实现了上述技术.实验结果表明:对于扩展了增量语法描述的 TPC-H 测试集, HiveInc 相对于优化前,可以获得平均 2.93 倍、最高 5.78 倍的加速;与经典优化技术 IncMR, DryadInc 相比,分别可以获得 1.69 和 1.61 倍的加速.

本文第 1 节介绍相关工作的优势与不足.第 2 节介绍本文设计的增量描述语法以及语义规则指导的增量优化技术的处理流程.第 3 节介绍语义指导的增量优化规则以及基于该规则设计的增量查询计划生成算法.第 4 节介绍包括增量描述语法解析、增量相关部分识别、拆分操作符的代价选择在内的其他技术细节.第 5 节介绍基于 Hive 的原型实现.第 6 节分析实验数据.第 7 节总结全文.

1 相关工作

目前,针对大数据查询的增量优化研究工作,按照实现层次的不同可以分为两大类:(1) 基于执行引擎实现的增量查询优化技术,代表性的工作有 CBP^[7], IncMR^[8], Incoop^[9], DryadInc^[10]; (2) 基于 DAG 形式查询 workflow (图中节点代表一个可以在分布式计算框架上执行的任务)实现的增量优化技术,如 Nectar^[11], Nova^[12].

CBP^[7]将数据处理流程看作由一系列有状态的操作符(stateful operator)组成的 DAG.每个操作符都记录状态,新数据到来以后用操作符对应的函数将状态和新数据合并,并更新状态,状态可以理解成操作符的历史数据.更新状态的算法由用户实现.以图 2 所示的查询为例,用户需要实现 lineitem 表的新数据与其他表进行连接的算法,以及新的连接中间结果与历史中间结果的合并算法.

IncMR^[8]和 Incoop^[9]都基于 Hadoop^[13]实现,支持针对 MapReduce 的增量优化. IncMR 的实现方法是:记录每一个 map 任务的输出,当新数据到来后,只需启动用于处理新数据的 map 任务,然后将新 map 任务的输出和已存储的 map 输出一起作为 reduce 任务的输入. IncMR 会尽量将 reduce 任务调度在有历史结果的节点上,以减少数据迁移的代价. Incoop 将 reduce 任务组织为一棵名为 contraction tree 的二叉树的形式.树的根节点对应 reduce 函数,中间节点对应 combiner 函数,叶节点对应 map 任务的输出.为实现增量优化, Incoop 需要存储每个 MapReduce 作业的 map 输出以及 contraction tree 上每一个中间节点的计算结果.以图 2 所示的查询为例,该查询会被 Hive 划分成 8 个 MapReduce 作业.为了实现增量优化, IncMR 和 Incoop 需要存储其中每个作业的 map

输出。

DryadInc^[10]基于 Dryad^[14]实现,输入都是 DAG 形式的分布式任务图。DryadInc 支持两种增量优化处理办法:IDE 和 MER。IDE 采用启发式的历史数据存储策略:如果图中某个阶段的所有进程都受新输入数据的影响,则存储该阶段的输入;重用时,选择图中最大的并且已经存储了历史输出的子 DAG。MER 由用户指定存储/重用位置,以及重用位置上合并历史数据和新输入数据中间结果的函数。具体到图 2 所示的查询,最理想的存储位置是聚合操作的输入,但 IDE 会选择存储第 1 个连接操作(lineitem 为其直接输入)的输入,导致后续的 6 个连接操作都需要重复处理历史数据。

Nectar^[11]基于 DryadLINQ^[15]实现,增量作用在查询 DAG 图,基于代价选择对查询的优化改写方式。对于一个查询,Nectar 首先尝试存储查询树的所有前缀子树的中间结果以及生成该中间结果所需的时间;新数据到来后,从查询树的最大前缀子树 t (即查询树本身)开始,寻找是否已经存了 t 作用在当前数据集 D 或者 D 的某个子集上的历史结果,若有则停止遍历,若没有则迭代分析 t 的最大前缀子树,直至收敛。对于同一个查询,Nectar 可能会找出多种对历史结果的重用方法,可能是重用小数据集的长路径计算结果,也可能是重用大数据集的短路径计算结果。Nectar 根据所记录的生成各个中间结果所需的时间,选择节省时间最多的历史结果重用方法。Nectar 同样存在无差别存储所有子树的计算结果的问题。Nectar 的优势在于支持包括周期性增量查询之外的多种查询场景,如查询 DAG 图之间的数据复用、移动窗口查询,等。

Nova^[12]是一个抽象层次更高的增量式工作流管理(workflow manager)系统。工作流是由任务(task)和管道(channel)构成的一个 DAG,由用户指定每一个任务处理的输入数据的类型(new,all)、对输入数据的处理方式(non-incremental,stateless incremental,stateful incremental)以及输出数据的类型(base,delta)。此外,用户还要指定每个管道调用的数据处理函数(merge,chain,diff)。

此外,还有一些其他增量处理技术(如 Percolator^[6],HaLoop^[16],Trill^[17],Redoop^[18]),与本文关注的应用场景不同,故本节不做介绍。

表 1 从用户透明性、重用粒度选择和历史数据存储选择这 3 方面总结了上述工作,从中可以看出,它们或者需要用户来选择历史数据中间结果存储点、设计增量处理逻辑,增加了普通用户的使用困难;或者会存储多余的历史数据中间结果,从而带来不必要的开销。本文针对上述问题,设计了一套以查询操作符语义为指导的增量优化规则。优化规则遵循语义等价、推迟合并和同位存储这 3 条原则,以确保在保持查询语义的前提下,最大限度地重用历史数据的查询结果。并且设计了一个用于控制历史结果的存储规模的代价模型,降低优化过程中的额外开销,进一步提高优化收益。优化过程对用户透明。

Table 1 Comparison of related works

表 1 相关工作对比

	用户透明性	重用粒度	历史结果存储选择规则
CBP ^[7]	否	mr 任务	用户指定
IncMR ^[8]	是	Map 输出	所有 Map 输出
Incoop ^[9]	是	Map+Reduce 输出	所有 Map 输出
DryadInc ^[10]	是	子 DAG	所有 combiner 输出
Nectar ^[11]	是	子查询	用户指定 DAG/最大公共子 DAG 输出
Nova ^[12]	否	类 Sql 任务/mr 任务	根据 cost 选择
本文方法	是	子查询	用户指定

2 系统概述

针对相关研究工作在用户透明性和优化效果方面的不足,本节为用户设计了一种增量描述语法以及一种以查询语义为指导的自动优化周期性增量查询的方法。

2.1 增量描述语法

增量描述语法设计成 SQL 语句注释的形式。用户可以在注释中指定周期性查询的开始时间、间隔周期和

结束时间.描述增量语义的注释既可以添加在查询语句的结束位置,也可以添加在某一数据表尾部.当添加在某一数据表尾部时(即图 1 所示的情况),表示仅将该数据表作为增量优化算法的输入对象;否则,表示将查询中所有包含周期性增量数据的表都作为增量优化算法的输入对象.

以下注释写法均符合增量描述语法.

- (1) /*+INCREMENTAL(after 2014/3/4,12:5:20 INTERVAL 3/M)*/,代表从 2014 年 3 月 4 日的 12 点 5 分 20s 开始,每隔 3 分钟执行一遍查询,结束时间没有设置,代表查询将周期性重复,直到用户手动终止.
- (2) /*+INCREMENTAL(2014/3/4-2014/3/5)*/,代表从 2014 年 3 月 4 日 0 点起,到 2014 年 3 月 5 日 0 点,期间按照默认间隔周期,重复执行查询.
- (3) /*+INCREMENTAL(after 2014/3/4)*/,代表从 2014 年 3 月 4 日 0 点起,到用户手动终止前,按照默认间隔周期,重复执行查询.

2.2 增量优化流程

本文设计的增量优化流程如算法 1 所示(如图 3 所示).

算法 1. 增量优化主算法 $IncTransform(HiveQL)$.

```

1  SyntaxTree=compile(HiveQL);
2  StartTime,Interval,StopTime=getIncInfo(SyntaxTree);
3  noIncList,IP=IPIdentify(SyntaxTree);
4  IncPlan=IncPlanGenerate(IP,Rules);
5  execute(noIncList);
6  timer.schedule(
7    {findIncFiles(IncPlan);
8     execute(IncPlan);},
9    StartTime,Interval,StopTime);

```

Fig.3 Algorithm for Hive incremental computation

图 3 增量优化算法

算法 1 包括 4 个主要步骤(原理图如图 4 所示).

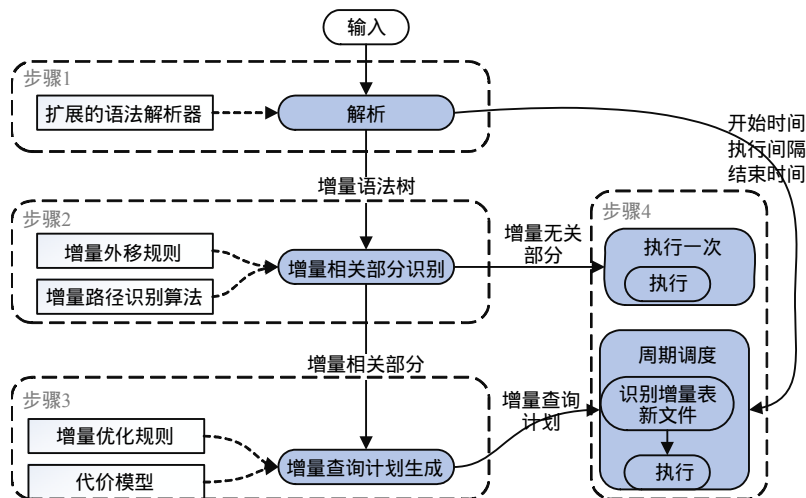


Fig.4 Architecture

图 4 原理图

1) 增量查询语法解析(算法 1 第 1 行、第 2 行).

本步骤扩展了主流分布式查询系统的语法前端,用以解析用户采用第 2.1 节所介绍的增量描述语法所描述

的信息.解析得到的周期性查询起止时间与间隔周期以及增量优化算法的输入表信息(简称增量表),将附着在解析得到的查询语法树上,并随着优化的进行,传递给新生成的查询中间表示,分别用于增量查询计划的周期性调度(步骤 4)和增量相关部分的识别(步骤 2).使用扩展的语法解析器解析 SQL 查询,得到标注了增量表的查询树和增量信息,增量信息包括查询开始执行时间 *StartTime*、查询执行间隔 *Interval*、查询结束时间 *StopTime*.比如,图 2 所示查询中的增量语法描述经扩展的语法前端解析后,得到的抽象语法树如图 5 所示,图中使用缩进表示父子关系.

```

1.  (TOK_JOIN
2.    (TOK_TABREF
3.      (TOK_TABNAME(lineitem))
4.      (TOK_INCRE
5.        (TOK_STARTTIME(TOK_DATETIME(TOK_DATE(1995)/(/)(1)/(/)(1))))
6.        (TOK_INTERVAL(90)(days))
7.        (l))
8.      (TOK_TABREF(TOK_TABNAME(part))(p))
9.      (and
10.        (=.(p)(p_partkey)).((l))(l_partkey)))
11.      (=.(p)(p_type))('ECONOMY ANODIZED STEEL'))))
    
```

Fig.5 Example for syntax tree (incremental algorithm description part)
图 5 语法树(增量语法描述部分)示例

图 2 所示查询的整个语法树可以简化地表示为图 6(1)所示的树型结构,其中,增量表 *l* 用方框标出(限于篇幅,图中对语法树和查询计划都进行了简化.语法树的叶子节点代表输入表,其余节点代表对应基本逻辑操作符的语法标识符.查询计划上的节点代表基本查询操作符,叶子节点代表扫描对应表的操作符).

2) 增量相关部分识别(算法 1 第 3 行).

本步骤以步骤 1 生成的包含增量信息的查询语法树为输入,将查询语法树划分成增量数据流可以达到的增量相关部分和不受增量数据影响的增量无关部分.其中,仅增量相关部分由 1 条以上(含 1 条)的增量路径构成.增量路径是语法树上以增量表标识符为起点的路径,路径的长度为路径包含的能够转化成基本逻辑操作符的标识符的数量.增量相关部分的规模为各条增量路径的长度之和,如果两条增量路径之间有重叠,则重叠部分的长度仅计算 1 次.增量相关部分需要周期性重复执行,增量无关部分可以只执行 1 次,把执行结果存储起来,供后续查询复用.可见,即便不对增量相关部分的进行增量优化,仅扩大增量无关部分,也可以降低查询代价.因此,本步骤在识别增量路径的过程中,根据“增量外移规则”对查询语法树进行等价变换.增量相关部分将交由步骤 3 处理,增量无关部分将从语法树中提取出来,转换成独立的查询,查询结果写入增量无关部分存储表中.比如,图 6(1)中查询语法树,经过等价变换,增量相关部分的规模从 10 缩小到 6(图 6(2)中黑色部分),交由步骤 3 处理;增量无关部分(图中灰色子树)转换成两个独立的查询计划,直接交步骤 4 处理.

3) 增量查询计划的生成(算法 1 第 4 行).

本步骤的输入包括:步骤 2 识别的增量相关部分、根据查询操作符的计算语义和输出语义设计的状态转换表形式的增量优化规则以及用于判断如何生成拆分操作符的代价模型.增量优化规则是本文方法的核心,决定增量路径上历史数据中间结果的存储位置和合并位置.增量查询规则的设计遵循语义等价、延迟存储和同位存储这 3 条原则.规则细节见第 3 节.增量查询计划生成算法根据增量优化规则选定的存储位置、合并位置以及存储对象,结合对代价模型公式以及物理查询任务划分点的分析,确定最终的存储、合并位置,然后为存储位置上的操作符插入用于存储规则所指定的输入的拆分操作符子树;将合并位置上操作符替换成用于读取已存储的历史数据中间结果,并将它们与新增数据当前处理结果合并的合并操作符子树,生成增量查询计划.图 6(3)为增量相关部分经增量查询计划生成算法处理后生成的增量查询计划.算法根据增量查询规则选择 *TOK_GBY_sum* 为存储兼合并位置,生成物理查询计划后表示为 *GBY_sumOp.GBY_sumOp* 的拆分操作符子树有两种可选形式,即图 6(3)中标注的形式、形式.算法将根据代价模型(第 4.3 节)选择其中存储、合并开销较小的形式.

4) 查询计划的调度执行(算法 1 第 5 行~第 9 行).

本步骤的输入包括:步骤 2 生成的增量无关部分的查询计划、步骤 3 生成的增量查询计划以及步骤 1 解析得到增量信息.增量无关执行一次并存储结果,待增量查询计划使用(算法第 5 行).增量查询计划依照用户指定的开始时间、执行间隔和结束时间周期性调度执行.每次执行前,对于需要扫描增量表的查询任务,识别增量表对应路径下上次执行后新增的文件列表.修改对应物理任务的输入,令其只扫描新增文件列表(算法第 7 行).然后,将物理查询计划提交到分布式查询引擎中执行.如图 6(4)所示的中间表示代表查询计划,右边两棵树是增量无关部分的查询计划,只需执行 1 次,查询结果分别写入临时表 *noInc0* 和 *noInc1* 中,原查询计划中的对应位置修改为对表 *noInc0* 和 *noInc1* 的扫描,左边树是增量查询计划,要周期调度执行,每次执行时只扫描增量表 *l* 新增数据,记为 Δl .

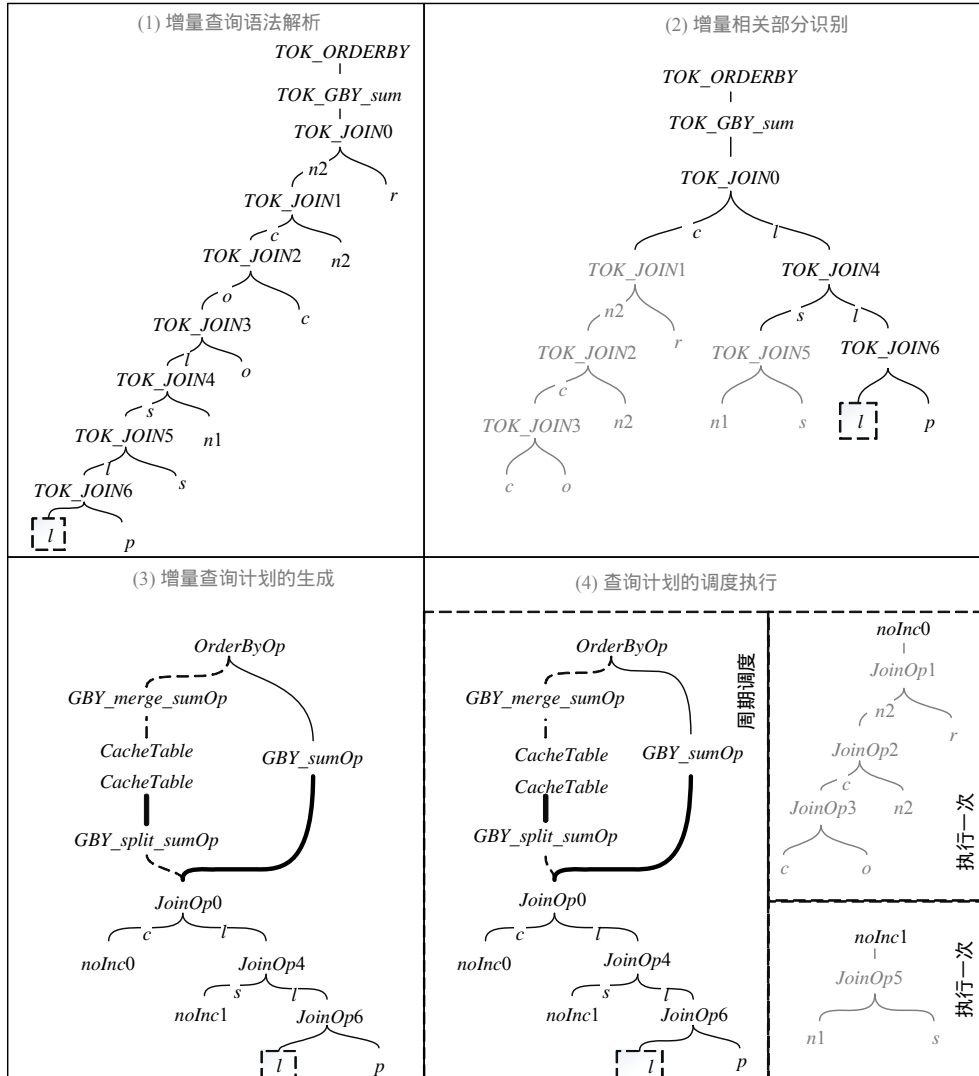


Fig.6 Example for incremental optimization process

图 6 增量优化过程示例

3 增量优化规则与算法

本节介绍语义指导的增量优化规则(第 3.1 节)以及基于规则设计的增量查询计划生成算法(第 3.2 节).

增量优化规则的输入是查询语法树上的增量相关部分,输出为历史查询中间结果的存储位置和合并位置.增量查询计划生成算法将根据增量优化规则的输出拆分查询语法树:在存储位置对应的操作符与其待存储的输入之间插入用于拆分原始查询语法树、存储历史查询中间结果的操作符子树(下文统称拆分操作符);将合并位置上操作符的输入子树替换成用于读取、处理历史数据查询中间结果,再合并历史数据中间结果与新增数据当前处理结果的操作符子树(下文统称合并操作符),生成增量查询计划.

3.1 增量优化规则

定义. 增量路径是查询语法树上一条以增量表标识符为起点、以语法树根节点为终点的路径.

一棵查询语法树上可能包含多条增量路径,增量路径之间允许有重叠.由同一棵查询语法树上的增量路径构成的集合称为增量相关部分.

增量路径的长度为路径上包含的能够被转化成基本逻辑操作符的标识符的数量.基本逻辑操作符是指完成单一特定功能的操作符.表 2 所示为 Hive 中部分基本逻辑操作符与语法树上标识符之间的对应关系(一条增量路径上的所有 TOK_WHERE 标识符在计算路径长度时只算 1 次).这些基本逻辑操作符在转化成物理查询计划时,可以由一个独立的 map 或 reduce 阶段或者一个独立的 MapReduce 作业来完成.

Table 2 Part of Hive logical operators and corresponding identifiers in syntax tree
表 2 Hive 部分基本逻辑操作符与语法树上标识符的对应关系

基本逻辑操作符	语法树标识符	基本逻辑操作符	语法树标识符
TableScan operator	TOK_TABREF	GroupBy operator	TOK_GROUPBY
Filter operator	TOK_WHERE	OrderBy operator	TOK_ORDERBY
Join operator	TOK_JOIN	Limit operator	TOK_LIMIT
Select operator	TOK_SELECT	RollUp operator	TOK_ROLLUP_GROUPBY
-	-	Cube operator	TOK_CUBE_GROUPBY

增量相关部分的规模为它所包含各条增量路径的长度之和.如果两条增量路径有重叠,则重叠部分的长度仅计算 1 次.

增量优化规则与增量查询计划生成算法的设计遵循语义等价、推迟合并和同位存储这 3 条基本原则.语义等价是优化规则的设计基础,即确保插入的拆分操作符和合并操作符不会违反原始查询的语义.推迟合并和同位存储的目的是在维持语义等价的前提下提高优化收益.在选择存储/合并位置时,一方面令存储位置与合并位置重叠,并尽量推迟它们在增量路径上的位置,可以避免对历史数据的重复处理开销,提高优化收益;另一方面,存储/合并历史数据中间结果可能引入额外开销(比如新的物理查询任务),为了降低或避免额外开销,应尽量将存储/合并位置选择与物理查询任务划分点相同的位置上.

设计的依据是各语法标识符对应的基本逻辑操作符(下文简称操作符)的计算语义和输出语义.其中,操作符的计算语义决定当选择该操作符作为存储/合并位置时,应插入何种拆分/合并操作符,以维持查询语义等价;操作符的输出语义决定是否必须选择该操作符作为存储/合并位置,即如果提前或推迟存储/合并位置,是否会影响到查询语义的维护.表 3 按照操作符的计算语义,将常用操作符分成 I 型操作符和 II 型操作符.其中, D 代表上一次查询结束后,增量表中存储的数据; ΔD 代表从上一次查询结束到本次查询开始之前新增加的数据.

Table 3 Classification of operators

表 3 操作符分类

op 类型	分类依据	特点	代表性操作符
I 型	$OP_I(D \cup \Delta D) = OP_I(D) \cup OP_I(\Delta D)$	历史数据中间结果和新增数据中间结果可以通过 UNION 操作符合并	TS, FIL, Join, MiJoin
II 型	$OP_{II}(D \cup \Delta D) \neq OP_{II}(D) \cup OP_{II}(\Delta D)$	历史数据中间结果和新增数据中间结果不能通过 UNION 操作符合并	GBY, ORDERBY, LIMIT

I 型操作符满足对集合并的分配律 $OP_I(D \cup \Delta D) = OP_I(D) \cup OP_I(\Delta D)$.代表性的 I 型操作符有扫描操作符(TableScanOperator,记作 TS)、过滤操作符(FilterOperator,记作 FIL)和连接操作符(JoinOperator)等.注意:连接操

作符存在单输入增量(记作 Join)和多输入增量(记作 MiJoin)两种情况,都属于 I 型操作符.当选择 I 型操作符作为合并位置时,通过 UNION 操作符就可以完成对新旧数据查询中间结果的合并(记作 UNION 合并).以两个输入都有增量(D_1, D_2 代表历史输入, $\Delta D_1, \Delta D_2$ 代表新增输入)的 MiJoin 为例,有:

$$MiJoin(D_1 \cup \Delta D_1, D_2 \cup \Delta D_2) = Join(D_1, D_2) \cup Join(D_1, \Delta D_2) \cup Join(\Delta D_1, D_2) \cup Join(\Delta D_1, \Delta D_2).$$

除 I 型操作符外的操作符统称 II 型操作符.代表性的 II 型操作符有聚合操作符(GroupByOperator, 记作 GBY)、排序操作符(OrderByOperator, 记作 ORDERBY)、取前 x 个操作符(LimitOperator, 记作 $LIMIT_x$).如果选择一个 II 型操作符作为合并位置,必须使用专门的合并操作符(见后文表 6)合并新旧数据的查询中间结果(记作非 UNION 合并).

表 4 定义了操作符的 3 种输出语义.输出语义由输出数据的完整性以及输出数据中包含的来自历史数据的中间结果的可重用性两种性质构成.一个操作符的输出语义由该操作符的计算语义以及孩子操作符的输出语义共同决定.

Table 4 Output semantics of operators

表 4 操作符的输出语义

输出语义	完整性	可重用性	满足性质的操作符举例
01	N	Y	不包含合并操作符的增量路径前缀上的操作符
10	Y	N	增量路径上位于非 UNION 合并操作符之后的单输入 I 型操作符序列中的操作符
11	Y	Y	增量路径上位于 UNION 合并操作符之后的单输入 I 型操作符序列中的操作符

完整性代表该操作符的输出包含了后续操作符在本轮查询中需要处理的所有输入.输入被增量描述语法描述为增量表的操作符,输出不具有完整性.如果一个操作符的任一孩子操作符的输出不具有完整性,则该操作符的输出也不具有完整性.将某一操作符的输出性质从不完整转化为完整的唯一途径是选择该操作符作为合并位置.

可重用性代表输出中由历史数据产生的部分并未与由新增数据产生的部分融合,而是可以独立地被后续操作符所使用.I 型操作符满足对集合的分配律,由新增数据产生的输出可以作为一个独立的部分追加在历史输出之后.如果一个 I 型操作符的所有孩子操作符的输出都具有可重用性,则该操作符本身也具有输出可重用性;如果其中任一孩子操作符不具有输出可重用性,则该 I 型操作符的输出也不可重用.增量路径上的 II 型操作符以及位于非 UNION 合并之后的操作符的输出都不具有可重用性.

综上,增量优化规则的核心思路可以归纳为:对于一个具有输出可重用性的操作符,合并位置允许推迟或提前,在尚未知晓物理查询任务划分位置的情况下,优选推迟;对于一个输出不可重用的操作符,如果其输出不完整,则必须被选合并位置;令存储位置与合并位置重合.

基于操作符的计算语义和输出语义设计的增量优化规则见表 5.

Table 5 Storage position and merger position choice rule in state-transition form of incremental optimization

表 5 状态转换表形式的增量优化存储、合并位置选择规则

增量优化规则	右孩子的输出语义		
	01	10	11
左孩子的输出语义	01	11($LS+RS+M$)	10($LS+M$)
	10	10($RS+M$)	10
	11	01	10
			[I]01/[II]10($LS+M$)
			10
			11

优化规则在自底向上遍历查询语法树的过程中实施.对于每个操作符,根据其左、右孩子操作符输出的完整性和可重用性以及操作符本身所属的类型,决定是否选择其作为存储位置或重用位置以及操作符的输出语义.对于只有 1 个孩子的操作符,默认该孩子为左孩子,将不存在的右孩子的输出语义预设为完整的、可重用的(即状态 11).规则设计的一个特例是,选择所有孩子的输出都不具有完整性但满足可重用性的 MiJoin 作为存储兼合并位置.以两个孩子的 MiJoin 为例,根据语义,存储、合并位置都可以继续推迟,推迟后虽然可以节省重复执行 $Join(D_1, D_2)$ 的代价,但也会增加合并代价:合并过程中的 $Join(\Delta D_1, D_2)$ 和 $Join(D_1, \Delta D_2)$ 在执行时会各生成一个

新的物理查询任务.尤其是在增量路径上有多个连续的 MiJoin 时,推迟合并可能导致变换后的查询计划及其复杂,不利于查询效率和语义维护.除特例外,规则默认选择的都是增量路径上能够维持优化前后语义等价的最近存储、合并位置.在物理查询任务生成过程中,规则选定的位置可以提前到与最近一个任务划分点重合.

表 5 中的 L 表示当前操作符的左输入, R 表示当前操作符的右输入, S 表示存储位置, M 表示合并位置, $[I]$ 和 $[II]$ 分别代表当前操作符的类型.比如,

- 当左孩子的输出语义为 01、右孩子的输出语义为 01 时,处理规则“11($LS+RS+M$)”代表以当前操作符为存储、合并位置,存储对象是当前操作符的左输入和右输入,设置当前操作符输出语义为 11.
- 当一个操作符的左孩子的输出语义为 01、右孩子的输出语义为 11 时,需要进一步考虑操作符类型:若当前操作符是 I 型,则不进行任何存储或合并变换,仅将操作符的输出语义置为 01;若为 II 型,则以当前操作符为存储、合并位置,存储对象是当前操作符的左输入,设置当前操作符的输出语义为 11.

下一节将介绍如何根据规则,选择存储、合并位置,执行增量优化变换.

3.2 增量查询计划的生成算法

图 7 所示为增量查询计划生成算法的伪码.

```

算法 2. 增量计划生成算法 IncPlanGenerate(SyntaxTree,IP,Rules).
1.  (SplitOPs,CachePoses)=FindCachePosition(SyntaxTree,IP,Rules)
2.  SplitedSyntaxTreeList=doSplitMerge(SplitOPs,Rule.function)
3.  for (AST:SplitedSyntaxTreeList+SyntaxTree)
4.    opTree=GenIncLogicalPlan(AST,cachePoses); /*传递 AST 存储合并点到 opTree*/
5.    taskTree=GenIncPhysicalPlan(opTree); /*提前存储合并点到最近的任务划分点*/
6.    IncPlan.add(taskTree);
7.  endfor
8.
9.  调用函数 FindCachePosition(SyntaxTree,IP,Rules)
10. 按照拓扑序、自底向上遍历查询语法树
11.  If (OP.childNum==0) then
12.    DataStatus(OP)=OP∈IP ? 01:11;
13.  else
14.    leftChildStatus=DataStatus(OP.leftChild);
15.    rightChildStatus=(OP.childNum==1)? 11:DataStatus(OP.rightChild);
16.    Rule=LookupRules(Rules.position,OP.type,leftChildStatus,rightChildStatus);
17.    If (Rule.store!=null & Rule.merge!=null) then
18.      If (CostModel(OP)=true) then
19.        SplitOPs+=OP;
20.      else CachePoses+=(Rule.merge,Rule.store); /*记录存储合并点*/
21.      endif
22.    endif
23.  endif
24.
25. 调用函数:doSplitMerge(SplitOPs,Rule.function)
26.  for(OP in SplitOPs)
27.    (SplitOP,mergeOP)=LookupRules(Rule.function,OP);
28.    FSOP=makeParentOP(splitOP,“存储表名”);
29.    TSOP=makeChildOP(mergeOP,“存储表名”);
30.    child=OP.child; parent=OP.parent;
31.    child.parent=splitOP; parent.child=mergeOP;
32.    SplitedSyntaxTreeList.add(FSOP);
33.  Endfor

```

Fig.7 Algorithm for incremental query plan generation

图 7 增量查询计划生成算法

算法以增量相关部分(IP)和上一节设计的增量优化规则(rules)为输入.增量优化规则包括表 5 所示的状态转换表(*Rules.position*)和表 6 所示的 II 型操作符的拆分合并操作符形式(*Rules.function*).算法主体为负责识别存储/合并位置的 *FindCachePosition* 函数以及负责插入拆分、合并操作符的 *doSplitMerge* 函数.

函数 *FindCachePosition* 按照拓扑序、自底向上遍历查询语法树(*SyntaxTree*).对于作为叶子节点的扫描操

作符,如果其被标记为增量输入,则将操作符的输出数据语义初始化为不完整的、可重用的(01),其余扫描操作符的输出语义置为完整的、可重用的(11)(算法 2 第 11 行、第 12 行).对于非叶子节点操作符,读取其左、右孩子节点操作符的输出语义.对于只有单个孩子节点的操作符,设置其右孩子的输出为完整的、可重用的(11)(算法 2 第 14 行、第 15 行).用操作符类型(I 型、II 型)和左、右孩子操作符输出语义查寻状态转换表,获得处理规则(rule).规则由存储位置(Rule.store)、合并位置(Rule.merge)和规则实施后输出语义(Rule.dataStatus)构成(算法 2 第 16 行).如果存储/合并位置非空,则根据代价模型(第 4.3 节)选择拆分、合并操作符(算法 2 第 18 行).表 6 列出了部分典型 II 型操作符应使用的拆分、合并操作符.其中,拆分操作符有两种:一种是只带存储功能的朴素拆分操作符,另一种是附带合成器功能的拆分操作符.合成器的作用类似于 MapReduce 中的 combine 函数,可以缩小待存储数据的规模.两种拆分操作符对应的合并操作符也可能不同.以 GBY 为例,朴素拆分操作符是 FS,对应的合并操作符是 $GBY_{key,func}$,带合成器的拆分操作符是 $GBY_{key,combiner.FS}$,对应的合并操作符是 $GBY_{key,merge}$.其中,合成器对应的 combiner 函数以及合并操作符中的 merge 函数由 GBY 的聚合函数 func 来决定,详见表 7.代价模型(第 4.3 节)将决定究竟选择哪一种拆分操作符以及它对应的合并操作符.如果选择结果为使用带合成器的拆分操作符,则后续不再调整存储/合并位置,将该位置以及需要存储的对象传递给 doSplitMerge 函数(算法 2 第 19 行).如果选择结果为使用朴素的拆分操作符,则将存储/合并位置以及待存储的对象缓存起来,待物理查询任务划分完毕后,再根据任务划分点决定是否将存储/合并位置提前(算法 2 第 20 行).

Table 6 Split operator and merger operator of operators II representatives

表 6 典型 II 型操作符的拆分、合并操作符

II 型操作符	拆分操作符	合并操作符
$GBY_{key,func}$	$[GBY_{key,combiner.}]FS$	$GBY_{key,func}(GBY_{key,merge})$
$Limit_x$	$[Limit_x.]FS$	$Limit_x$
OrderBy	FS	OrderBy
$RollUp_{key,func}$	$[RollUp_{key,combiner.}]FS$	$RollUp_{key,func}(RollUp_{key,merge})$
$Cube_{key,func}$	$[Cube_{key,combiner.}]FS$	$Cube_{key,func}(Cube_{key,merge})$

Table 7 Combiner function and merger function of some aggregate functions

表 7 部分聚合函数的 combiner,merge 函数

聚合函数 func	combiner 函数	merge 函数
$Sum(col)$	$Sum(col)$ as col1	$Sum(col1)$
$Count(col)$	$Count(col)$ as col1	$Sum(col1)$
$Avg(col)$	$Sum(col)$ as col1, $Count(col)$ as col2	$Sum(col1)/Sum(col2)$
$Max(col)$	$Max(col)$ as col1	$Max(col1)$
$Min(col)$	$Min(col)$ as col1	$Min(col1)$
$Distinct(col)$	$Distinct(col)$ as col1	$Distinct(col1)$

doSplitMerge 函数对每一个选定的存储/合并位置,将语法树从该位置断开:在存储位置对应的操作符与作为存储对象的输入之间插入拆分操作符,拆分操作符的输出写入到指定存储表;将合并位置上的操作符替换成合并操作符,再为合并操作符添加一个用于扫描指定存储表的孩子操作符(算法 2 第 27 行~第 32 行).对于变换产生的所有语法树,分别将它们转化成逻辑查询计划(算法 2 第 4 行),转化过程中,将缓存的(即还有待调整的)存储/合并位置映射到逻辑查询计划中的基本逻辑操作符节点上,再在生成物理查询任务的过程中,将可提前的存储/合并位置提前到最近的任务划分点(算法 2 第 5 行),输出优化后的增量查询计划.

图 8(1)所示为图 6(3)中增量查询计划的生成过程:首先,初始化叶子节点的输出语义, $noInc1, p, noInc0$ 是非增量表,初始化状态 11, l 是增量表,初始化为 01.当遍历到 GBY_sumOp 时,其左孩子输出语义为 01,不存在的右孩子的输出语义为 11,对应的处理规则为“10($LS+M$)”.如果代价模型判断结果为使用附带合成器的拆分操作符,则采用图中标注为 的变换方式;否则,采用图中标注为 的变换方式.

算法 2 同样适用于包含多个增量表的查询.图 8(2)所示查询包含两个增量表($l1, l2$),经算法 2 处理后选择了两个存储兼合并位置: $GBY_countOp$ 和 $JoinOp0$.其中, $JoinOp0$ 存储其右输入; $GBY_countOp$ 的拆分、合并操作符也有两种选择,图中只画出使用朴素拆分操作符的情况.

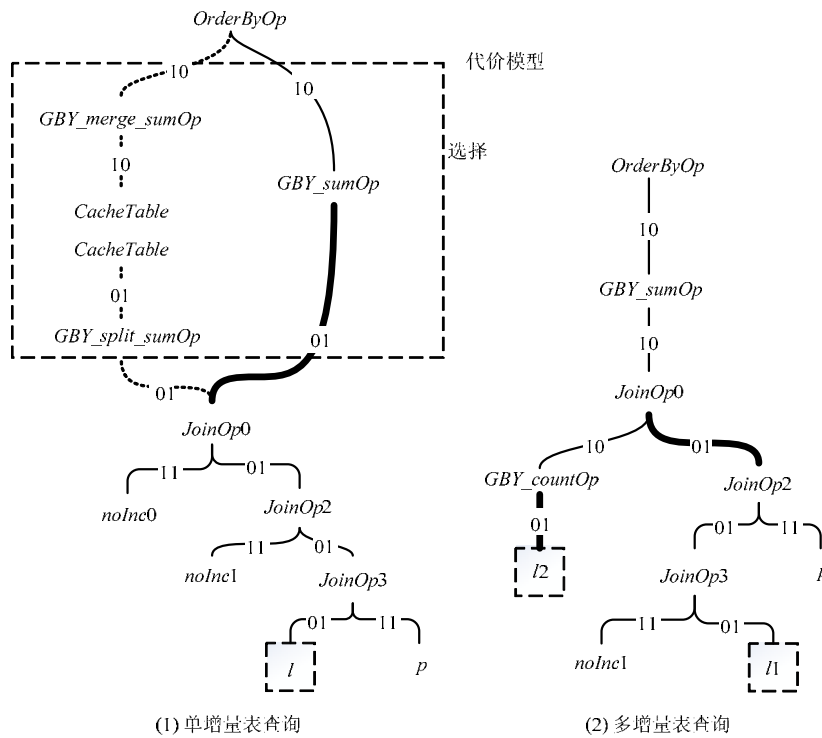


Fig.8 Example for incremental query plan generation
图 8 增量查询计划生成示例

4 技术细节

本节介绍语义规则指导的增量优化方法中,除增量优化规则之外的技术细节,包括增量描述语法解析、增量相关部分的识别和代价模型设计.

4.1 增量描述语法解析

为了解析第 2.1 节所扩展的语法表述,我们对语法解析器扩展 4 条解析规则.新增解析规则见表 8.

Table 8 Parsing rules of incremental description syntax
表 8 增量描述语法解析规则

	解析规则	HiveQL 语法示例	解析生成的语法树子树
1	tabName (incrClause)? →^(TOK_TABREF tabName incrClause?)	page /*+INCREMENTAL(string)*/ o	TOK_TABREF TOK_TABNAME(page) /*+INCREMENTAL(string)*/O
2	/*+INCREMENTAL(string)*/ →^(TOK_INCRE string)	/*+INCREMENTAL (string)*/	TOK_INCRE String
3	Stimeetime →^(TOK_STARTTIME stime) ^(TOK_STOPTIME etime)	2014/3/4-2014/3/5	TOK_STARTTIME 2014/3/4 TOK_STOPTIME 2014/3/5
4	after stime (INTERVAL num/unit)? →(INTERVAL!=NULL)? ^(TOK_STARTTIME stime) ^(TOK_INTERVAL num unit) →^(TOK_STARTTIME stime)	after 2014/3/4,12:5:20 INTERVAL 3/M	TOK_STARTTIME 2014/3/4,12:5:20 TOK_INTERVAL 3M
		after 2014/3/4	TOK_STARTTIME 2014/3/4

表中第 1 列是解析规则,第 2 列是该规则可处理的 HiveQL 语法示例,第 3 列是第 2 列的示例经过解析规则处理后得到的语法树子树(缩进代表标识符间的父子关系).解析规则 1、规则 2 用于解析增量描述语法中指定的增量表,规则 3、规则 4 用于解析增量查询的起始时间、结束时间和周期.第 3 列中的黑体标识符为增量描述语法引入的新标识符.后续算法将通过识别这些新标识符来获得对应的信息.

4.2 增量相关部分的识别

本节介绍增量相关部分的识别算法,以及如何在识别过程中,根据增量外移规则,扩大增量无关部分.

增量外移规则适用于增量路径上由满足交换律和结合律的多目操作符构成的路径前缀,最典型的是由 Join 操作符(即语法树上 TOK_JOIN 标识符)构成的增量路径前缀(Join 操作符满足 $Join(a,b)=Join(b,a)$ 和 $Join(a,Join(b,c))=Join(Join(a,b),c)$,其中 a,b,c 代表 Join 的输入).增量外移规则规定:对于两个相邻的 Join(分别记作 InnerJoin 和 OuterJoin,InnerJoin 是 OuterJoin 的左孩子),如果满足:1) InnerJoin 有且只有 1 个孩子的输出不具有完整的;2) OuterJoin 的右孩子具有完整性;3) OuterJoin 的左连接条件只来自于 InnerJoin 具有输出完整性的孩子这 3 个条件,则可以将查询语法树上 InnerJoin 的不具备输出完整性的输入子树与 OuterJoin 的右子树交换,缩短增量路径,扩大增量无关部分.

增量路径识别算法伪码如图 9 所示,算法的输入是查询语法树 SyntaxTree,输出包括一个增量相关部分(记作 IP)和一系列增量无关部分的根节点(保存在 noIncList 中).算法按照拓扑序、自底向上遍历查询语法树(SyntaxTree):将扫描增量表的(包含 TOK_INCRE)标识符加入 IP,其余表扫描标识符加入 noIncList(算法 3 第 1 行、第 2 行);对于只有 1 个孩子的标识符,其归属与孩子的归属相同,但由于 noIncList 只保存增量无关部分的根节点,所以当将一个标识符加入 noIncList 时,要将它的孩子从 nonIncList 中删除(算法 3 第 3 行、第 4 行);对于有两个孩子的标识符,如果两个孩子都属于 IP,则将其本身也加入 IP(算法 3 第 6 行、第 7 行);如果两个孩子都属于 noIncList,则用其本身替换 noIncList 中的左、右孩子(算法 3 第 8 行、第 9 行);如果两个孩子中的一个属于 IP 增量路径,且当前标识符为 TOK_JOIN,则进一步判断当前标识符与它的父亲是否满足增量外移规则必须满足的 3 个条件:若满足,则执行交换并更新 noIncList;若不满足,则更新 IP(算法 3 第 10 行~第 13 行).完成遍历后,将 noIncList 中的节点引导的增量无关子树从查询语法树中提取出来,剩下的就是增量相关部分.

```

算法 3. 增量相关部分识别算法 IPIdentify(SyntaxTree)
从叶子节点出发,按照拓扑序,遍历 SyntaxTree 上的每个标识符 t
1.  if (t.childNum=0) then
2.     (t 扫描增量输入)?(IP=IP+t):(noIncList=noIncList+t);
3.  else if (t.childNum=1) then
4.     (child∈IP)?(IP=IP+t):(noIncList=noIncListchild+t);
5.  else if (t.childNum=2) then
6.     if (Lchild∈IP & Rchild∈IP) then
7.         IP=IP+t;
8.     else if (Lchild∉IP & Rchild∉IP) then
9.         noIncList=noIncListLchildRchild+t;
10.    else
11.        child1=(Lchild∈IP)?Lchild:Rchild;
12.        child2=(Lchild∈IP)?Rchild:Lchild;
13.        sat=outShiftCondition(child1);
14.        (sat=true)?(noIncList=noIncListchild2+t):(IP=IP+t);
15.    endif;

```

Fig.9 Algorithm for incremental path recognition

图 9 增量路径识别算法

4.3 拆分操作符的选择

算法 2 在根据增量优化规则选择存储位置后,对于某些 II 型操作符提供了两种可选的拆分操作符:朴素的拆分操作符和附带合成器的拆分操作符.使用带合成器拆分操作符的优势是控制需要存储的历史数据中间结果的规模,但缺点是在被转化成物理查询任务时可能引入一个额外的执行合成操作的查询任务或者一个 map/

reduce 阶段.使用朴素的拆分操作符可以在物理查询任务生成后,根据任务划分点调整存储、合并位置,因此不会引入额外的优化开销.但是当所存储数据到达一定规模时,有可能增加合并操作符的扫描负担.综上,当读取未经合成的历史数据的开销超过因附带合成器的拆分操作符执行合成操作带来的额外开销时,选择附带合成器的拆分操作符能够获得更高的优化收益;反之,可以继续使用朴素拆分操作符.

公式(1)为拆分操作符的代价选择公式.

$$\frac{S(M + \Delta M)}{mDSz} - \frac{S(\Delta M)}{mDSz} - \frac{S(M' + \Delta M')}{mDSz} > \Gamma \tag{1}$$

公式左部的第 1 项代表合并操作符处理未经合成的历史数据和本次新增数据的开销,该开销主要取决于合并操作符的输入数据的规模(包括已存储的历史中间结果 M 的规模和本轮新增数据产生 ΔM 的规模)以及系统中可以同时处理的最大数据规模 $mDSz$. $mDSz$ 由系统中可以同时运行的最大任务数量($maxTaskNum$)和每个任务可处理的数据规模($splitSz$)决定. $mDSz$ 的计算方法以及公式(1)中其他符号的含义和计算方法见表 9.

Table 9 Meaning and computing method of symbols in cost formula

表 9 代价公式中各符号的含义及计算方法

符号	含义	计算方法
α	增量表中本次新增的数据量与历史数据量的比值	分别统计增量表目录下新增文件和旧文件的规模后相除
$S(M)$	上一轮查询中合并操作符的输入数据规模	运行时统计
$S(Merge(M))$	上一轮查询中合并操作符的输出数据规模	运行时统计
$mDSz$	集群同一时间能够处理的最大数据规模	查看系统配置中节点数($numOfNode$)、每节点可用内存($memPerNode$)、可用核数($corePerNode$)、每个任务所需的内存($memPerJvm$)以及每个任务处理的数据规模($splitSz$): $mDSz = splitSz \times maxTaskNum, maxTaskNum = numOfNode \times \min(corePerNode, memPerNode / memPerJvm)$

公式(1)左部的第 2 项代表对本次新增数据执行合成操作的开销,第 3 项代表合并操作符处理合成后的历史数据(M')和本次新增数据(ΔM)的开销,计算方法同第 1 项.左部计算结果代表使用附带合成器的拆分操作符所能带来的收益.但在从朴素拆分操作符转变成选用附带合成器的拆分操作符的过程中也会引入额外的开销,即历史数据 M 需要被合成 M' .这部分开销会均摊到以后的每个增量周期,由公式右部的 Γ 表示. Γ 可以由用户设置,默认值为 0.公式(1)成立时,选用附带合成器的拆分操作符可以获得比朴素拆分操作符更高的优化收益.

公式(1)的最佳应用时机是每轮增量处理之前,但公式中 ΔM 的规模只有在本轮查询执行后才能获得, M' 和 $\Delta M'$ 的规模只有在选择并执行了附带合成器的拆分操作符之后才能获得.对此,我们默认在首轮查询中选用朴素拆分操作符,在之后的每个查询周期,分别用上周期中合并操作符的输入数据规模 $S(M)$ 和输出规模 $S(Merge(M))$ 来估算本轮查询可能生成的 ΔM 和 $\Delta M'$ 规模,得到的估算公式如公式(2)所示.

$$\frac{((1 + \alpha) \times S(M))}{mDSz} - \frac{(\alpha \times S(M))}{mDSz} - \frac{(2 \times S(Merge(M)))}{mDSz} > 0 \tag{2}$$

一旦选择使用附带合成器的拆分操作符,后续的每轮增量处理就将沿用这一处理方式,不再重新判断.

5 实现细节

我们以 Apache Hive^[19]为基础实现了本文方法的原型系统 HiveInc(<https://github.com/acthires/HiveInc>).图 10 所示为 HiveInc 的原理图.图中的右半部分为 Hive 的固有逻辑,用于接收 HiveQL 查询(CliDriver),并依次将查询转化为抽象语法树(ParseDriver)、查询块(SemanticAnalyzer.doPhase1)、操作树(SemanticAnalyzer.genPlan)以及由 MapReduce 作业构成的 DAG 形式的物理查询计划(MapReduceCompiler).图中的左半部分为 HiveInc 为支持增量处理所做的扩展:虚线箭头表示对固有逻辑的扩展,实线箭头代表输入输出流(灰色实线箭头表示 Hive 原本的输入输出流).其中,增量查询计划生成算法分 3 部分实现.

- 1) 识别增量相关部分以及根据状态转换表形式的增量优化规则识别存储、合并位置实现在 AST 上.在

识别增量相关部分的过程中,我们只将能够生成独立分布式任务的增量无关部分(TOK_JOIN 或 TOK_QUERY 引导的增量无关子树)从查询语法树中提取出来.

- 2) 根据拆分合并函数表和代价模型选择并插入拆分操作符、合并操作符实现在 AST 转化成 QB 的过程中,如果用户选择使用代价模型,doSplitMerge 为每个存储、合并位置生成两种执行计划:一种使用朴素拆分方法,另一种使用附带合成器的拆分方法.默认使用前者,每一轮增量处理过程中,根据对上一轮运行结果的统计更新代价公式,如果判断为真,则改为使用后者.用户也可以选择不使用代价模型而是默认使用某一种拆分、合并操作符的处理方式.
- 3) 根据物理查询任务的划分点调整存储、合并位置实现在物理查询计划的生成过程中.

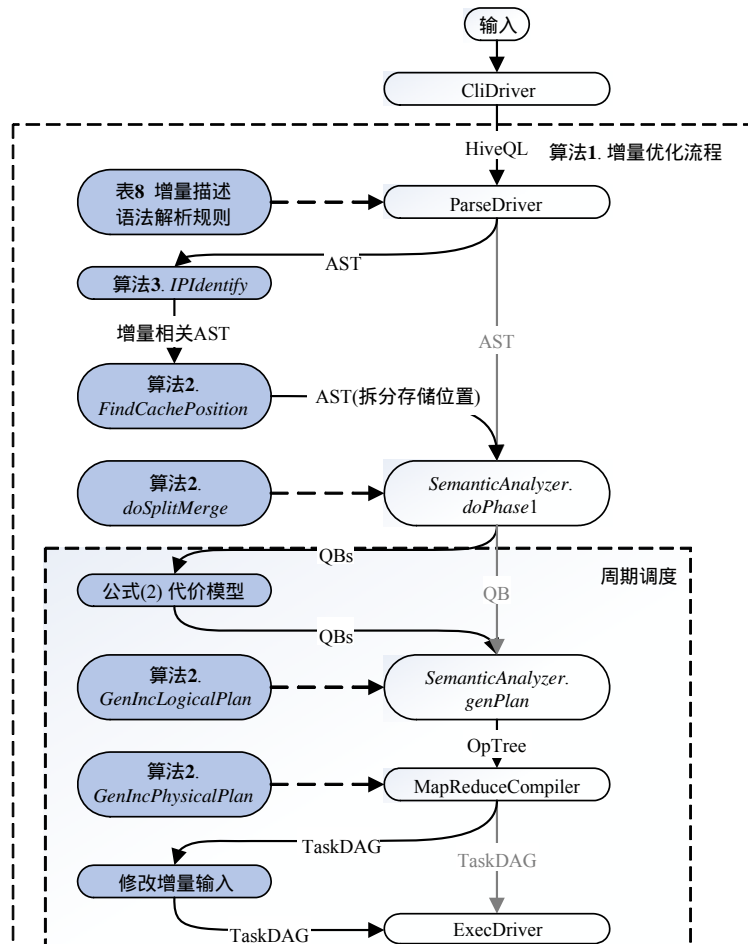


Fig.10 Implementation of HiveInc

图 10 HiveInc 实现

6 实验分析

实验环境采用由 4 台曙光 I840-G20 服务器搭建的集群.每个节点上配有 4 个主频为 2.0GHz 的 8 核 Intel E7 处理器,4 条 8GB 的 DDR4 内存.其中,原型可用的核数为每节点 8 个,可用内存为每节点 12GB.集群上安装的用于存储查询数据、执行查询任务的 Hadoop^[13]版本为 Apache Hadoop-2.6.2.测试用例集(<https://github.com/acthires/HiveInc/tree/master/TestCases>)基于标准测试集 TPC-H^[20]设计.TPC-H 是学术界和工业界普遍使用的评

测数据仓库系统性能的测试集,根据真实的工业使用环境设计,包含 8 张基本关系表和 22 个查询用例.实验选用 TPC-H 中除 Q15(Q15 必须写成两条查询语句的形式,但目前我们的原型不支持多语句间的增量优化,故暂不使用)外的 21 个查询用例,数据表由自带的 dbgen 数据生成工具生成.我们按照第 2.1 节介绍的增量描述语法,为每个测试用例添加了描述增量表、增量起始时间和间隔周期的注释.其中,默认选择规模最大的表为增量表.如果最大的表被多次使用,则该查询为一个包含多表增量的查询.各个查询用例的编号、用途以及涉及的 SQL 操作符见表 10.

Table 10 Features and functions of experimental queries

表 10 实验用例查询特点及用途

编号	涉及的 SQL 操作符	用途
Q1	投影、过滤、分组聚合(求和)、排序	对于每种返回标记、状态组合,统计其数量、折扣前价格、折扣后价格、加税后价格的总和及平均值
Q2	投影、连接、求最小值、排序、前 100 个	查找欧洲价格最低的黄铜供应商,并按照账户余额降序排列,显示前 100 个
Q3	投影、连接、过滤、分组聚合(求和)、排序、前 10 个	查找客户为建筑领域的、还没有被运输的价值最大的 10 个订单
Q4	投影、连接、过滤、分组聚合(计数)、排序、去重	按照优先级,分组统计至少包括 1 个零件的收货时间晚于承诺日期的订单数量
Q5	投影、连接、分组聚合(求和)、排序	对于亚洲每个国家,统计供货、消费都在本国的收益总量
Q6	投影、过滤、求和	统计订购的零件数量不低于 24、折扣区间为[0.05,0.06]的订单带来的收益
Q7	投影、连接、分组聚合(求和)、排序	对于从法国到德国和从德国到法国的运单,分别统计其每年的收益
Q8	投影、连接、分组聚合(求和)、排序、Case 子句	统计巴西的电镀钢在美洲所占的市场份额
Q9	投影、连接、分组聚合(求和)、排序	统计名称为“%green%”的零件在每个国家、每年的利润
Q10	投影、连接、分组聚合(求和)、排序、前 20 个	统计退单总金额前 20 个客户的信息
Q11	投影、连接、过滤、分组聚合(求和)、排序	对于德国的大于所有零件总价值的 0.000 02 倍的各种零件,分别统计其价值
Q12	投影、连接、过滤、分组聚合(求和)、排序、Case 子句	对于高、低两个优先级,统计船运和邮寄两种方式分别造成延迟交付的订单数
Q13	投影、左外连接、分组聚合(计数)、排序	统计每个订单数对应的顾客数量
Q14	投影、连接、求和、Case 子句	统计促销零件的收益占总收益的比例
Q16	投影、连接、分组聚合(计数)、排序	对于品牌不是“Brand#45”、类型不是“MEDIUM POLISHED%”、零件尺寸为指定大小的每种商品,统计可提供该种商品且没有顾客投诉的供应商的个数
Q17	投影、连接、过滤、分组聚合(求和、求平均)	统计除“Brand#23”品牌之外、容器为“MED BOX”、数量低于平均数量 0.2 倍的零件,平均每年损失的收益
Q18	投影、连接、分组聚合(求和)、排序、前 N 个	统计包括 300 件以上零件,且订单总额前 100 的客户和订单信息
Q19	投影、连接、过滤、求和	统计特定品牌、包装、大小、运输方式的零件按照打折价出售的总收益
Q20	投影、连接、过滤、分组(求和)、去重、排序	统计加拿大可以提供商品名称为“forest%”,存货量大于 1994 年总出货量一半的供货商名字和地址
Q21	投影、连接、过滤、分组(计数、求最大)、右外连接、排序、前 N 个	查找由多个零件组成的订单中唯一一个延迟交付且来自沙特阿拉伯的供应商,统计其延迟交付的订单数,按照延迟订单数从大到小排序,显示前 100 个
Q22	投影、右外连接、连接、分组(计数、求和)、求平均、排序	对于特定 7 个国家,统计每个国家的账户余额大于平均值,且一直没有下订单的用户数,以及这些用户的账户总余额

6.1 优化分析

图 11 所示为 HiveInc 原型对初始数据 20GB,经过 9 个增量周期,每个周期增加数据规模与增量表的初始规模相等的测试集的优化加速比.对于测试集中的每个查询,图 11 分别展示了 10 次查询的平均加速比,以及当历史数据与新增数据规模比值分别呈 1 比 1、2 比 1、4 比 1 和 8 比 1 时的加速比.其中,所有 21 个查询经过 10 次增量的平均加速比为 2.93,单个查询的平均加速比最高可达 5.78(Q9);单次增量的平均加速比随着历史数据与新增数据比值的提高而提高,最高可达 10.29(Q19 的第 8 个增量周期).

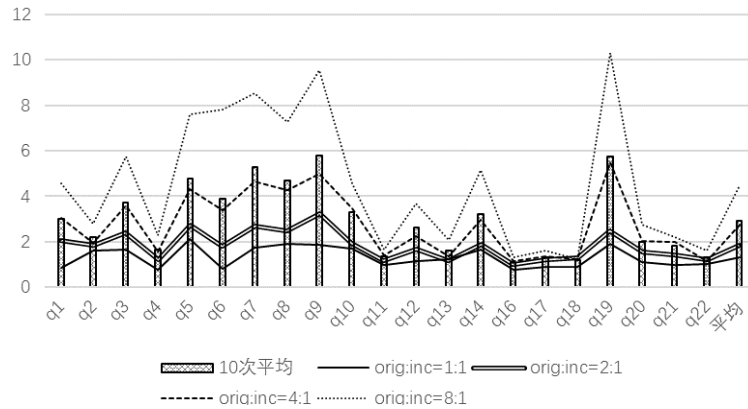


Fig.11 Speed-Up ration of hiveinc when incremental data of queries is constant

图 11 HiveInc 处理增量规模恒定查询的加速比概览

根据优化前后 10 次查询需要处理的数据规模,在假定对历史数据没有任何冗余处理的前提下,计算出的理想加速比为 5.5.但图 11 中,q9,q19 的平均加速比都显著高于理想加速比,q7 的平均加速比接近理想加速比.造成上述现象的原因如下:一方面,这 3 个查询本身具有增量路径长、路径上生成的物理查询任务的输入数据量大的特点,增量对查询时间的影响大,优化余地也大;另一方面,HiveInc 不仅能够提取出部分耗时但与增量无关的查询操作,而且存储、合并位置之后不再包含耗时的查询操作,存储的数据规模也始终保持在一个较低且恒定的水平.以 q9 为例,优化前的 q9 在每轮增量查询时会生成 7 个物理查询任务,其中 6 个与增量有关.优化前,查询时间的增长与输入数据量的增长几乎呈线性(图 12(1)).优化后,与增量无关,用于连接 nation 表和 supplier 表的部分在语法树阶段就被单独提取出来,且只执行 1 次;且由于存储、合并位置选在所有连接完成之后的 GBY 处,前 4 个连接任务都只需要处理新增数据;需要存储的数据量也一直维持在较低的水平,代价判断结果始终选择朴素的拆分操作符.从图 12(1)可以看出,经 HiveInc 优化后,q9 每个增量周期的查询执行时间维持 1 000s 左右.

对于 q1,q3,q5,q6,q8,q10,q12,q14 这 8 个查询,HiveInc 选择的存储、合并位置之后同样不包耗时的查询操作,但由于这些查询的增量路径偏短,优化前查询时间受增量数据的影响不如前述的 3 个查询明显,因此,平均加速比也略低于前 3 个查询,维持在 3 倍~4 倍之间.以 q8 为例(图 12(2)),虽然优化后每轮增量查询的执行时间维持在 362s 左右,但由于优化前每轮增量查询的时间较首轮查询仅增加 43%,理论上的平均加速比最高也只能到 2.36 倍.但从图 11 可以看出,当历史数据与增量数据的规模比达到 8:1 时,q8 单次优化加速为 7.28;由于优化后的查询时间几乎恒定,随着增量周期的增长,q8 的单次加速比和平均加速比都将稳固提高.

剩余的平均加速比低于 3 倍的查询又可以分为两类.

- 第 1 类包括 q4,q17,q18,q20,q21.对于这类查询,HiveInc 为了维护优化前后的语义等价而选择了 1 个或多个相对靠前的存储、合并位置,合并后仍然需要执行一些耗时的查询操作,导致优化后,查询时间仍然随数据量的增长而增长.以 q17 为例(如图 12(3)所示),其中包含两个增量表,分别被 GBY 和 Join 使用,GBY 的输出同时又是 Join 的输入.根据查询语义,Join 的输出不完整也不可重用,存储、合并位置不能再推迟.因此,HiveInc 选择 GBY 任务和 Join 任务的输入为存储、合并位置,存储对象包括 GBY 任务的输入、JOIN 任务除 GBY 输出外的另外两个输入.从第 1 个增量周期起,选择根据代价公式,选用附带合成器的拆分操作符,但生成的 4 个查询任务中仍然有 3 个需要重复处理历史数据.随着历史数据规模的增加,每轮增量的查询时间增加的时间约为首轮查询的 50%.
- 第 2 类查询包括剩余的 q2,q11,q13,q16,q22.这 5 个查询由于输入数据规模和增量表本身的规模都比较小,优化前后,系统资源都足以一次性处理每个查询任务的所有数据.以 q13 为例(如图 12(4)所示),优化前,每个增量周期,查询时间仅增加不到 20%;优化后,虽然每个增量周期的查询时间只增加 4%,但 10 次

查询的平均加速比只能达到 1.62.对于这类查询,只有在数据规模增加到一定程度,或系统资源受限时, HiveInc 才能发挥优化效果.

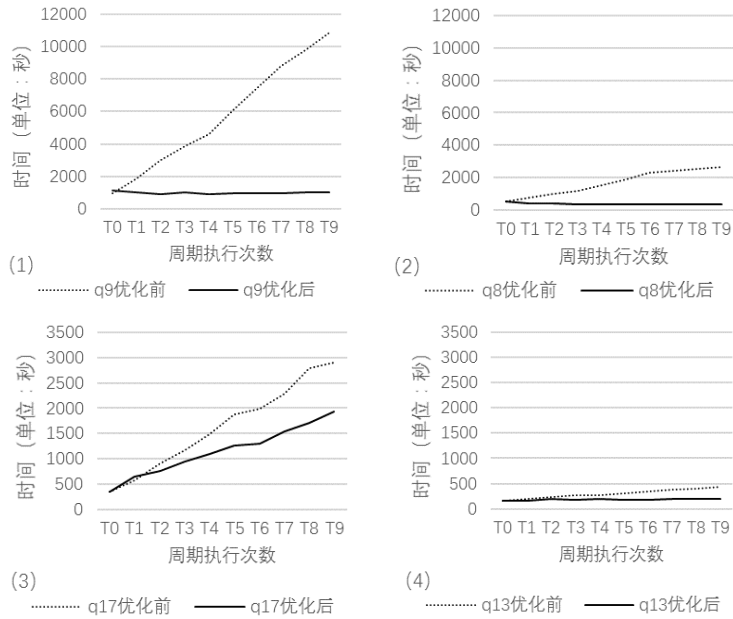


Fig.12 Execution analysis for various query examples

图 12 各类查询示例执行分析

图 13 对比了 HiveInc 使用代价模型选择拆分操作符(图中记作 HiveInc)与默认选用附带合成器的拆分操作符(图中记作合成器拆分)、默认选用朴素拆分操作符(图中记作朴素拆分)作用在上述测试集上的加速比.从中可见,两种默认选择方式相对于优化前所能获得的平均优化收益分别为使用代价模型收益的 95%和 88%.

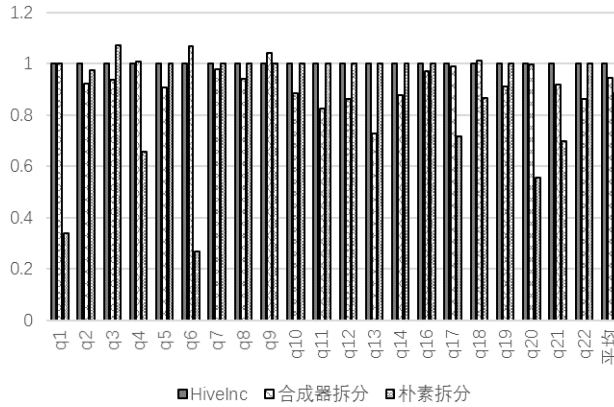


Fig.13 Speed-Up ration comparison of hiveinc, merger split, simple split

图 13 HiveInc、合成器拆分、朴素拆分的加速比对比

图 14 对比了 HiveInc 使用代价模型选择拆分操作符与默认选用附带合成器的拆分操作符时需要存储的历史数据中间结果规模.对于每个查询,代价模型在每个增量周期存储的历史数据中间结果平均为 1 177MB,为合成器拆分需要存储的规模(1385MB)的 85%.这是由于部分查询使用合成器拆分操作符后,存储中间结果的位置

与物理任务划分点不重合.因此,对于那些不需要从一开始就使用合成器缩小历史中间结果规模的查询,默认使用合成器拆分反而会额外存储更多的数据.

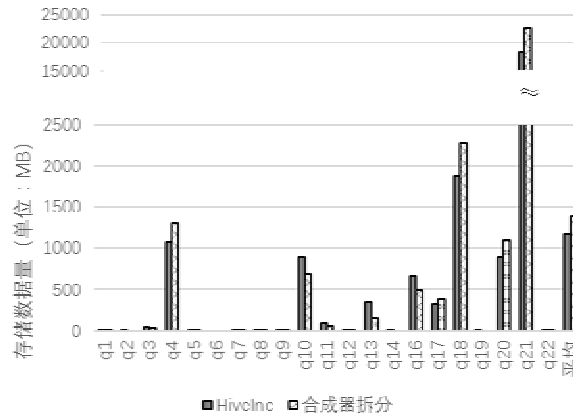


Fig.14 Data storage comparison of hiveinc, merger split
图 14 HiveInc 和合成器拆分需要存储的数据量对比

图 15 所示为 HiveInc 使用代价模型选择拆分操作符与默认选用朴素拆分操作符时合并操作符分别需要扫描数据规模.对于每个查询,使用代价模型后,合并操作符在每个增量周期需要扫描的平均数据规模为 1.17GB, 仅占朴素拆分(30.7GB)的 3.8%.可见,缩小合并操作符的扫描开销是代价模型相对朴素拆分操作符能够获得更高优化收益的一个重要原因.

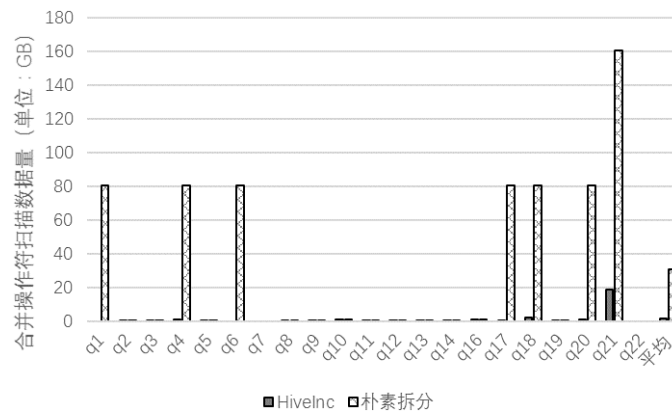


Fig.15 Merge-Op scan data comparison of hiveinc, simple split
图 15 HiveInc 和朴素拆分时合并操作符需要扫描的数据量对比

图 16 所示为 q21 在初始数据量为 10GB、9 个增量周期的数据量依次为 10G,20G,10G,25G,30G,20G,25G,20G,30G 时,HiveInc 使用代价模型选择拆分操作符、默认选用附带合成器拆分操作符、默认选用朴素拆分以及优化前的各个增量周期的执行时间,其中,合成器拆分、朴素拆分以及优化前的平均执行时间分别是 HiveInc 的 1.04,1.33,1.85 倍.HiveInc 从第 1 个增量周期(T1)开始选择使用合成器拆分.从图中可以看出,T1 以前,使用合成器拆分的执行时间高于朴素拆分;T1 以后,使用朴素拆分的执行时间迅速增加.可见,HiveInc 的代价模型选择是合理的.又由于使用代价模型需要存储的历史数据量比默认使用合成器拆分存储数据量小,因此,T1 以后 HiveInc 的执行时间也低于合成器拆分的执行时间.只有在 T1 当次执行时,由于要从朴素拆分为合成器拆分,HiveInc 的执行时间略高于其他方法,但这部分开销被以后每个增量周期的收益所抵消.

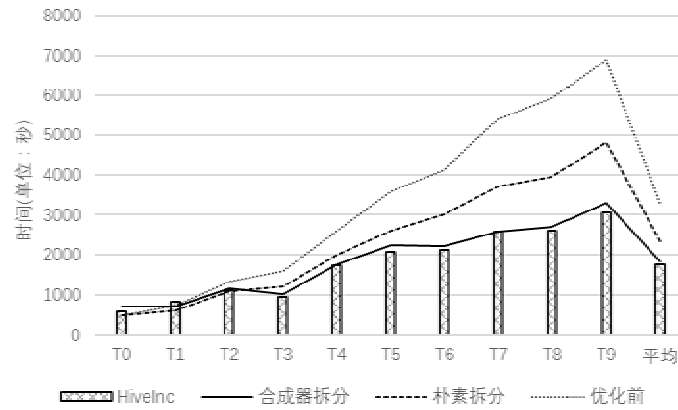


Fig.16 Optimization comparison when the incremental data is variable (q21)

图 16 增量数据规模可变时的优化效果对比(q21)

6.2 对比

本文模拟了 IncMR^[8]和 DryadInc^[10]的 IDE 策略的优化效果.仍选用初始数据 20GB,9 个增量周期,每个周期增加数据规模与增量表的初始规模相等的 TPC-H 测试集.图 17 所示为 HiveInc,IncMR 和 DryadInc 作用在 21 个查询上的平均加速比,以 HiveInc 的加速比为标准化基数.从中可见,IncMR 和 DryadInc 的平均优化加速分别只能达到 HiveInc 的 59%和 62%.对于 21 个查询中的 13 个,HiveInc 优化加速更高的原因在于,其选择的存储、合并位置都是最优的.而 IncMR 和 DryadInc 只有第 1 个查询任务的 Map 阶段可以只处理新增数据,其 Reduce 阶段以及后续每个查询任务都需要处理所有数据.对于剩余的 8 个查询(q4,q12,q14,q16,q17,q18,q20,q21),HiveInc 选择的存储、合并位置相对靠前,且只能选在查询任务的划分点处;IncMR 和 DryadInc 由于修改了查询执行引擎,可以重用查询任务内部部分 map tasks 的计算结果,优化收益更高.可见,如果将 HiveInc 和相关工作联用,可以优势互补,获得更高的优化收益.

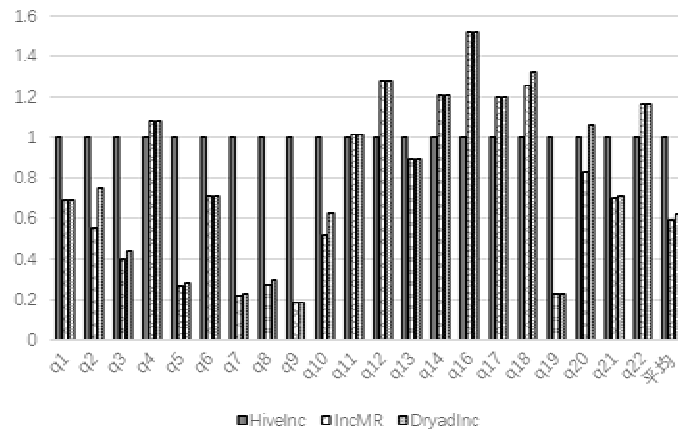


Fig.17 Speed-Up ratio comparison of HiveInc, IncMR and DryadInc(normalized to HiveInc)

图 17 HiveInc,IncMR 和 DryadInc 加速比对比(以 HiveInc 为标准化基数)

图 18 对比了 HiveInc,IncMR 和 DryadInc 对于每个查询,在每个增量周期,需要存储的历史数据中间结果的平均规模.从中可见,HiveInc 需要存储的数据量仅占 IncMR 的 5.5%,占 DryadInc 的 9.1%.这一方面受益于本文设计的增量优化规则可以找到查询相关部分中最优的历史中间结果存储点;另一方面,当历史数据规模太大时,代价模型会自动选择使用附带合成器的拆分操作符,缩小数据规模.

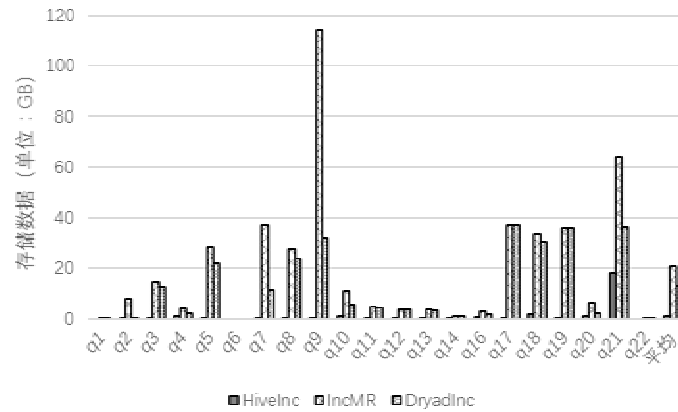


Fig.18 Data storage comparison of HiveInc, IncMR and DryadInc

图 18 HiveInc, IncMR 和 DryadInc 存储的数据量对比

7 总 结

本文针对数据仓库中固定表增量的周期性查询场景,设计并实现了一种语义指导的增量优化方法.实验结果表明,该方法相对于优化前,可以获得平均 2.93 倍、最高 5.78 倍的加速;与经典优化技术 IncMR, DryadInc 相比,分别可以获得 1.69 倍和 1.61 倍的加速.

目前, HiveInc 原型可以支持连接、聚合等主要的 SQL 操作符以及其他 Apache Hive 0.13 版本可以接收的 SQL 操作符,比如 ROLLUP, CUBE 等.我们正在将原型移植到最新版的 Hive 上,以便更加全面地支持各类 SQL 语句.本文的下一步工作还包括:完善对包含多条查询语句的查询用例的优化支持以及对多种增量场景的支持,比如移动窗口形式的增量优化场景.

References:

- [1] Facebook process more than 500TB data daily. 2012 (in Chinese). <http://tech.sina.com.cn/i/2012-08-23/10597538323.shtml>
- [2] Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P. Hive: A warehousing solution over a map-reduce framework. Proc. of the VLDB Endowment, 2013,2(2): 1626–1629. [doi: 10.14778/1687553.1687609]
- [3] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. Proc. of the Operating Systems Design and Implementation, 2004,51(1):107–113. [doi: 10.1145/1327452.1327492]
- [4] Zaharia M, Chowdhury M, Das T, Dave A, Ma J, Mccauley M. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proc. of the 9th USENIX Conf. on Networked Systems Design and Implementation, Vol.70. Vancouver: USENIX Association, 2012.
- [5] Chattopadhyay B, Lin L, Liu W, Mittal S, Aragona P, Lychagina V. Tenzing: A SQL implementation on the MapReduce framework. Proc. of the VLDB Endowment, 2011,4(12):1318–1327.
- [6] Peng D, Dabek F. Large-Scale incremental processing using distributed transactions and notifications. In: Proc. of the 9th USENIX Conf. on Operating Systems Design and Implementation. Vancouver: USENIX Association, 2010. 1–15.
- [7] Logothetis D, Olston C, Reed B, Webb KC, Yocum K. Stateful bulk processing for incremental analytics. In: Proc. of the 1st ACM Symp. on Cloud Computing. Indianapolis: ACM Press, 2010. 51–62. [doi: 10.1145/1807128.1807138]
- [8] Yan C, Yang X, Yu Z, Li M, Li X. IncMR: Incremental data processing based on MapReduce. In: Proc. of the 2012 IEEE 5th Int'l Conf. on Cloud Computing. IEEE Computer Society. 2012. 534–541. [doi: 10.1109/CLOUD.2012.67]
- [9] Bhatotia P, Wieder A, Rodrigues R, Acar UA, Pasquin R. Incoop: MapReduce for incremental computations. In: Proc. of the 2nd ACM Symp. on Cloud Computing. Cascais: ACM Press, 2011. 1–14. [doi: 10.1145/2038916.2038923]
- [10] Popa L, Budi M, Yu Y, Isard M. DryadInc: Reusing work in large-scale computations. In: Proc. of the 2009 Conf. on Hot Topics in Cloud Computing. San Diego: USENIX Association, 2009.

- [11] Gunda PK, Ravindranath L, Thekkath CA, Yu Y, Zhuang L. Nectar: Automatic management of data and computation in datacenters. In: Proc. of the 9th USENIX Conf. on Operating Systems Design and Implementation. Vancouver: USENIX Association, 2010. 1–8.
- [12] Olston C, Chiou G, Chitnis L, Liu F, Han Y, Larsson M. Nova: Continuous Pig/Hadoop workflows. In: Proc. of the 2011 ACM SIGMOD Int'l Conf. on Management of data. Athens: ACM Press, 2011. 1081–1090. [doi: 10.1145/1989323.1989439]
- [13] Apache. Hadoop: Open-Source implementation of MapReduce. 2016. <http://hadoop.apache.org/>
- [14] Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: Distributed data-parallel programs from sequential building blocks. In: Proc. of the 2nd ACM SIGOPS/ EuroSys European Conf. on Computer Systems 2007. Lisbon: ACM Press, 2007. 59–72. [doi: 10.1145/1272996.1273005]
- [15] Yu Y, Isard M, Fetterly D, Budiu M, Erlingsson Ú, Gunda PK, Currey J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In: Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation. San Diego: USENIX Association, 2008. 1–14.
- [16] Bu Y, Howe B, Balazinska M, Ernst MD. HaLoop: Efficient iterative data processing on large clusters. Proc. of the VLDB Endowment, 2010,3(1-2):285–296. [doi: 10.14778/1920841.1920881]
- [17] Chandramouli B, Goldstein J, Barnett M, Deline R, Fisher D, Platt JC. Trill: A high-performance incremental query processor for diverse analytics. Proc. of the VLDB Endowment, 2014,8(4):401–412. [doi: 10.14778/2735496.2735503]
- [18] Lei C, Zhuang Z, Rundensteiner EA, Eltabakh MY. Redoop infrastructure for recurring big data queries. Proc. of the VLDB Endowment, 2014,7(13):1589–1592. [doi: 10.14778/2733004.2733037]
- [19] Apache. Hive: A data warehouse software facilitates querying and managing large datasets residing in distributed storage. 2016. <http://hive.apache.org/>
- [20] TPC-H. 2016. <http://www.tpc.org/tpch/>

附中文参考文献:

- [1] Facebook 每天数据处理量超 500TB.2012. <http://tech.sina.com.cn/i/2012-08-23/10597538323.shtml>



康炎丽(1991 -),女,山西五台人,硕士,主要研究领域为程序分析,大数据查询优化.



王蕾(1989 -),男,博士生,主要研究领域为程序分析,程序优化,软件安全.



李丰(1985 -),女,博士,助理研究员,主要研究领域为程序分析,缺陷定位.