计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2020.20190563
57(2):318-332,2020

# 基于收益模型的 Spark SQL 数据重用机制

申毅杰　曾　丹　熊　劲
(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)
(中国科学院大学　北京　100049)
(shenyijie@ict.ac.cn)

# A Benefit Model Based Data Reuse Mechanism for Spark SQL

Shen Yijie, Zeng Dan, and Xiong Jin
(*State Key Laboratory of Computer Architecture* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)
(*University of Chinese Academy of Sciences*, *Beijing* 100049)

**Abstract**　Analyzing massive data to discover the potential values in them can bring great benefits. Spark is a widely used data analytics engine for large-scale data processing due to its good scalability and high performance. Spark SQL is the most commonly used programming interface for Spark. There are a lot of redundant computations in data analytic applications. Such redundancies not only waste system resources but also prolong the execution time of queries. However, current implementation of Spark SQL is not aware of redundant computations among data analytic queries, and hence cannot remove them. To address this issue, we present a benefit model based, fine-grained, automatic data reuse mechanism called Criss in this paper. Criss automatically identifies redundant computations among queries. Then it uses an I/O performance aware benefit model to automatically choose the operator results with the biggest benefit and cache these results using a hybrid storage consisting of both memory and HDD. Moreover, cache management and data reuse in Criss are partition-based instead of the whole result of an operator. Such fine-grained mechanism greatly improves query performance and storage utilization. We implement Criss in Spark SQL using modified TachyonFS for data caching. Our experiment results show that Criss outperforms Spark SQL by 40% to 68%.

**Key words**　data analytics; big data; Spark SQL; redundant computation; data reuse; benefit model

**摘　要**　通过数据分析发现海量数据中的潜在价值,能够带来巨大的收益.Spark 具有良好的系统扩展性与处理性能,因而被广泛运用于大数据分析.Spark SQL 是 Spark 最常用的编程接口.在数据分析应用中存在着大量的重复计算,这些重复计算不仅浪费系统资源,而且导致查询运行效率低.但是 Spark SQL 无法感知查询语句之间的重复计算.为此,提出了基于收益模型的、细粒度的自动数据重用机制 Criss 以减少重复计算.针对混合介质,提出了感知异构 I/O 性能的收益模型用于自动识别重用收益最大的算子计算结果,并采用 Partition 粒度的数据重用和缓存管理,以提高查询效率和缓存空间的利用率,充分发挥数据重用的优势.基于 Spark SQL 和 TachyonFS,实现了 Criss 系统.实验结果表明:Criss 的查询性能比原始 Spark SQL 提升了 46%～68%.

**收稿日期**:2019-08-16;**修回日期**:2019-11-11
**基金项目**:国家重点研发计划项目(2016YFB1000202);国家自然科学基金项目(61379042)
This work was supported by the National Key Research and Development Program (2016YFB1000202) and the National Natural Science Foundation of China (61379042).
**通信作者**:熊劲(xiongjin@ict.ac.cn)

**关键词** 数据分析;大数据;Spark SQL;重复计算;数据重用;收益模型

**中图法分类号** TP316.81.2

海量数据中蕴藏着巨大的价值.分析海量数据、挖掘其中的潜在价值,能够为企业带来巨大的收益.例如风险预测、精准营销、商品推荐等.数据分析的常用平台是传统数据库.然而,随着数据量的急剧增长,数据库已无法适应大数据时代的需求,大数据处理系统应运而生.近年来,Spark[1-3]系统被广泛地应用在生产环境中.截至 2019 年,Spark 最大的集群已有 8000 节点,单个 job 处理的数据量可达到数千万亿字节[4].在进行数据分析时,用户经常使用的接口是 Spark SQL[5].Spark SQL 接收用户输入的 SQL 语句,将其翻译成由 RDD(resilient distributed dataset)[3]构成的有向无环图(directed acyclic graph,DAG),提交给 Spark 执行引擎执行.

RDD[3]是 Spark 中对分布式数据集的基本抽象.Partition 是 RDD 中数据在集群节点间分布的最小粒度,也是 RDD 在集群上被并行处理的基本粒度.一个 Spark SQL 的查询,会被翻译成查询计划.查询计划是一棵算子树(operator tree).算子树上的每个节点是一个算子(operator),表示对数据的一种操作。比如,Filter 算子表示基于条件对数据筛选,Sort 算子表示基于属性对数据排序.算子树会被转化为由多个 RDD 组成的有向无环图(DAG).Spark 的 DAG Scheduler 按照相邻 RDD 间是否需要数据重分布(repartition),将 DAG 划分为一到多个 Stage.每个 Stage 包含对数据集的一部分操作.对每个 Stage,Spark 在集群中启动一到多个任务(task),每个任务处理一个 Partition 内的数据.

在实际的数据分析场景中,存在着很多重复计算.尤其是在交互式查询中,查询语句之间可能仅仅是参数不同,后续查询语句通常会根据之前的查询结果不断修正参数.Microsoft 研究表明[6],其日志挖掘应用中约有 30%的重复计算,其内部数据分析平台 SCOPE 上有 80%的重复计算[7].近年来,有很多研究工作都致力于减少重复计算[7-20],以减少资源的浪费,提升系统的性能.

减少重复计算通常有 2 种方式:合并公共计算和数据重用技术.合并公共计算[20-22]针对并发查询场景,将并发执行的查询任务之间相同的部分加以合并,使得相同部分只需计算一次,计算结果传递给各个查询.但它只适用于并发场景下,使用范围非常有限.数据重用技术则是将重复计算的结果保存下来,供后续计算重用.在选择数据保存时,有人工选择[23-25]和系统自动选择[9-14]2 种方式.相比之下,自动选择方式有着更大的灵活性与准确度.自动数据重用技术最初用于传统数据库,如 Vectorwise[9],MonetDB[10],Microsoft SQL Server[16]等.后来这一技术也被应用于大数据处理平台,如 PigReuse[20],CloudView[7]等.这些研究表明,自动数据重用技术能够带来性能收益.对于单条查询来说,可达到 10%~80%的性能提升[9],从整个系统来看,可达到 30%的性能提升[26].

Spark 也实现了 2 种数据重用:RDD 缓存和共享文件.这 2 种方式都依赖于用户选择缓存数据,属于人工选择缓存数据的方式,且在 Spark SQL 场景下有着一定的局限性:1)RDD 缓存只能用于相同 RDD 对象之间的数据重用.而用户每提交一条 SQL 语句,都会产生新的 RDD DAG,即使提交相同的 SQL 语句,也会产生不同的 RDD 对象,因此该方式对查询语句之间的重用计算无效.2)共享文件的缓存单位是 RDD 粒度,在数据规模较大时不能很好地利用缓存空间,且会导致频繁替换,缓存效率较低.

为了减少重复计算,本文将自动数据重用技术应用在 Spark SQL 中.自动数据重用技术需要解决 4 个关键问题:重复计算的识别、数据缓存位置、缓存数据的选择和数据重用粒度.在已有的工作中,这 4 个关键问题都有着不同的解决方案.

在重复计算的识别上,主要是通过查询匹配与改写来识别重复计算.查询匹配目前主要有 3 种方法:SQL 字符串匹配[27]、规范化查询模板[11,13,15]和基于算子的匹配[9,28].前 2 种方式都只能重用整条查询的执行结果,而基于算子的匹配能重用查询的中间结果,重用机会大.另外,不同的 SQL 语句可以表示相同的计算,而 SQL 字符串匹配不能识别出这类重复计算.规范化查询模板则由于表达能力的限制,导致能够识别出重复计算有限.本文采用基于算子的匹配,以识别出更多的重复计算.

在重用数据的缓存位置上,已有的工作都是使用单一介质(内存或者磁盘)[9-15].但是在大数据场景下,内存容量小、磁盘数据传输速度慢,都存在一定的局限性.本文采用混合介质存储以扬长避短,最大化系统的重用收益.

在缓存数据的选择上,基于规则的选择[13-14]考虑的因素较为单一,相比之下,基于收益模型的选择[9-12]能够更为准确地评估数据的重用价值.但是,现有的收益模型都没有考虑数据读写引入的时间开销,也没有考虑混合存储介质上数据速度的不同,因此现有的模型都不能准确地评估重用收益.本文提出了新的重用收益模型,不仅考虑了数据读写的时间开销,而且针对混合介质,考虑了在不同介质上数据读写时间的差异,能够更为准确地评估数据的重用收益.

在数据重用粒度上,传统数据库中多采用算子粒度重用[9-13].算子粒度重用是把算子执行的结果缓存下来,缓存替换时以一个算子的完整执行结果为粒度进行替换.在分布式场景下,算子的计算结果由分布于多台机器上的 Partition 组成.本文提出了 Partition 粒度的数据重用,即数据重用和缓存替换都是以 Partition 为粒度.在分布式场景下,所有机器上缓存空间不会同时耗尽.只要有一台机器的缓存耗尽,算子粒度就需要替换某个(或某些)算子的所有 Partition.而 Partition 粒度则只需替换缓存耗尽的那个机器上的 Partition.因此,相比于算子粒度,Partition 粒度的数据重用能减少不必要的替换,提高数据的缓存效率和缓存空间的利用率.

基于以上分析,针对 Spark SQL,本文提出了基于收益模型的自动数据重用机制.针对混合介质,提出了感知异构介质的收益模型用于自动识别重用收益大的数据,并采用细粒度的数据重用方式以提高数据的缓存效率及缓存空间的利用率,充分发挥数据重用的优势.基于 Spark SQL 和 TackyonFS,本文实现了具有数据重用功能的 Criss 系统,能够根据历史负载自动识别出重复计算,并基于收益模型选择重用收益大的数据自动缓存,供后续计算重用,提升系统的查询处理性能.实验结果表明,Criss 的查询性能比原始 Spark SQL 提升了 46%~68%.

## 1 基于收益模型的数据重用机制

数据重用的前提是识别出重复计算.在传统数据库中,基于算子的匹配与改写方案在重复计算的识别度、重用机会和匹配开销方面都有较好的表现,本文采用这种方案,将其应用在大数据场景下.

为了在有限空间中缓存重用价值大的数据,本文提出了针对混合介质的收益模型用于评估数据的重用收益.一方面,基于大数据场景,采用混合介质构建缓存空间,相比于内存,提供更多的容量;相比于磁盘,在一定程度上提高数据的读写速度,从而提高系统整体的性能收益.另一方面,针对混合介质,提出一种感知异构存储介质的收益模型用于选择重用数据以及缓存数据的管理.

在数据重用粒度上,目前的方案都是算子粒度的数据重用,然而,在分布式场景下,算子的计算结果由分布于多个机器的 Partition 组成,且这些 Partition 是并行处理的,这使得细粒度的数据重用成为可能,因此,本文提出了基于 Partition 粒度的数据缓存与重用策略,以提高缓存效率及缓存空间的利用率.

基于以上思路,本文设计了具有数据重用功能的查询处理系统,其系统架构如图 1 所示.在将查询语句翻译成查询计划(算子树)后,将查询计划与历史查询负载进行匹配以识别重复计算.而且,根据匹配结果,对查询计划进行修改.

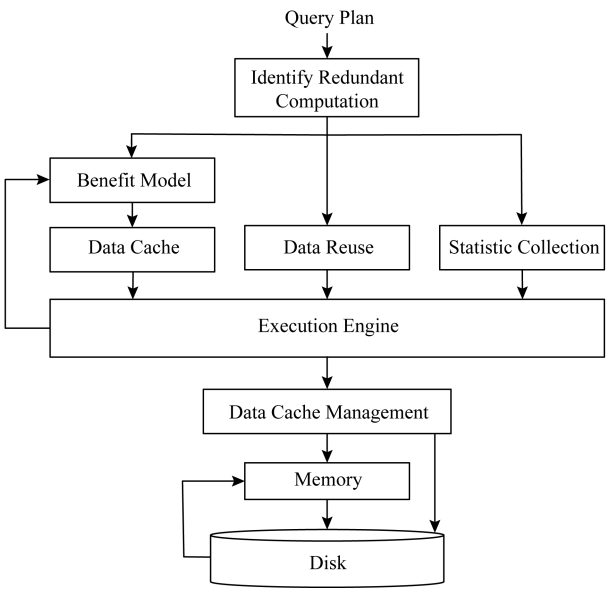



Fig. 1 System architecture

图 1 系统架构

对于查询计划中的每个算子,其修改体现在 3 个方面:

1) 是否自动缓存计算结果.根据收益模型计算重用收益,对于重用收益大的算子,在执行过程中自动缓存其结果.

2) 是否进行数据重用.若算子的计算结果已在缓存空间中,则修改其计算逻辑为从缓存空间中读取数据,而无需再进行重复的计算.

3) 是否需要收集统计信息.在选择计算结果进行缓存时,依据的标准是收益模型,而收益模型在

计算重用收益时需要获得算子的统计信息,因此需要在执行过程中添加统计信息收集功能.为了减少信息收集的开销,对于已经有统计信息的算子,在执行过程中不必收集.

对查询计划修改之后,将其提交给执行引擎.执行引擎根据情况进行相应的操作:或将算子的计算结果缓存起来,或直接读取缓存中的数据,或收集算子的统计信息.缓存空间采用内存和磁盘混合存储.数据在缓存空间足够时直接存放,优先存放在内存.空间不足时进行替换.对于磁盘中的数据,当其重用收益增大时,考虑将其迁入内存.为了在有限空间中缓存重用收益大的数据,还需要对缓存数据进行管理.

下面从重复计算的识别、收益模型(包括信息收集)、数据缓存、数据重用和缓存数据管理 5 个方面介绍本文的数据重用机制.

### 1.1 重复计算的识别

本文采用基于算子的匹配与改写方案识别重复计算,使用一个树结构(QTree)存储历史负载.QTree 中每个节点表示一个算子,包含一个全局 ID、算子的信息及其统计信息.其中,ID 与缓存空间中的数据一一对应,用于缓存数据定位;算子信息用于算子匹配;统计信息用于收益模型.

当系统新接收 1 条查询语句时,将其查询计划与 QTree 中保存的历史查询计划进行匹配,找出重复计算,进而找到可以重用的数据.匹配采用自底向上的方式进行,因为只有子算子都匹配成功,才能保证父算子的输入数据相同,这时再对父算子进行匹配.如果匹配成功,就认为该算子与 QTree 上算子的计算结果相同,它可以重用 QTree 上算子的计算结果.对于匹配成功的算子,只需更新相应算子的统计信息;对于匹配不成功的算子,将其插入 QTree 中.图 2 显示了系统在接收 2 条查询计划时 QTree 的变化情况.初始时 QTree 为空,只有 1 个根节点.用户提交第 1 条查询计划时,其算子都未在 QTree 上找到匹配算子,因此,将第 1 条查询计划的所有算子都插入 QTree.当用户提交第 2 条查询计划时,Project 和 Scan 算子都在 QTree 上找到匹配算子,而 Aggregate 未找到匹配算子,只需将它插入 QTree 中.

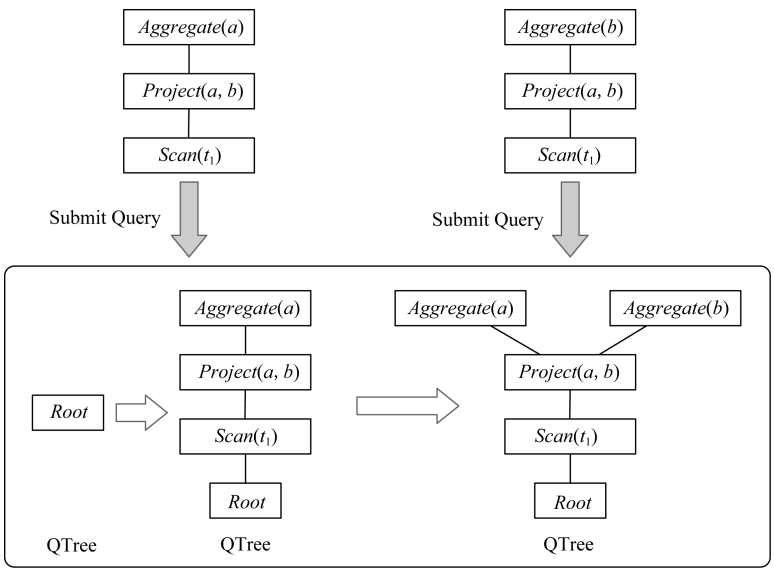



Fig. 2 An example of QTree structure

图 2 QTree 组织结构示例

### 1.2 收益模型

收益模型用于评估数据的重用收益.从理论上来讲,数据每被重用 1 次,获得的收益为算子的计算时间 $T_{\text{exec}}$ 与直接重用算子结果的时间 $T_{\text{load}}$ 的差值.在算子引用次数为 $ref$ 的情况下,缓存数据的重用收益为 $ref\times(T_{\text{exec}}-T_{\text{load}})$.在自动数据重用技术下,有重用价值的数据被自动缓存下来,因此缓存数据的重用收益还需减去数据写入缓存的时间开销 $T_{\text{store}}$,即 $ref\times(T_{\text{exec}}-T_{\text{load}})-T_{\text{store}}$.其中 $T_{\text{load}}$ 和 $T_{\text{store}}$ 分别为数据加载(从缓存中读取)和存储(写入缓存)的时间.它们可以通过算子结果的数据量 $size$ 除以存储介质带宽 $bw$ 近似计算得到,即 $T_{\text{load}}=T_{\text{store}}=size/bw$.而 $T_{\text{exec}}$ 需要在运行时收集算子的执行时间.另一方面,算子通常具有时间局部性,最近出现

过的算子越有可能再次出现,缓存其结果带来的重用收益就越大,即最近访问时间与重用收益成正相关.算子的最近访问时间 $recency$ 为当前时间与系统启动时间的差值.综上所述,本文构建的收益模型为

$$Benefit = recency \times (ref \times (T_{exec} - T_{load}) - T_{store}) = recency \times (ref \times (T_{exec} - size/bw) - size/bw).$$

收益模型的建立依赖于统计信息,因此系统还需要具有信息收集的功能.其中,算子的引用次数 $ref$ 和最近访问时间 $recency$ 在进行查询计划匹配时可以得到;算子的计算时间 $T_{exec}$ 和计算结果的数据量 $size$ 在运行时获取;存储带宽 $bw$ 可以根据系统环境获取.

由于本文使用混合介质构建缓存空间,根据存储介质的不同,在计算重用收益时带宽的设置不一样.系统中有 3 个地方需要计算重用收益:

1) 选择自动缓存结果的算子.根据统计信息计算重用收益,选择重用收益大的算子在此次执行过程中自动缓存其结果.由于计算收益时结果尚未缓存,无法得知将会被缓存在何种介质上.为了减少缓存开销,本文基于最坏的情况考虑,即假设数据会被缓存在磁盘,若此时数据的重用收益仍然很高,本文才会进行缓存,因此在选择自动缓存结果的算子时带宽设置为磁盘带宽.

2) 缓存空间不足进行替换时.本文采用基于收益模型的数据管理策略,优先缓存重用收益大的算子结果.在替换过程中需要计算重用收益.对于缓存空间中已有的数据,根据其具体的缓存位置设置带宽;对于当前待缓存的数据,根据其候选存储位置设置相应的带宽.

3) 磁盘数据迁入内存时.负载的变化会导致算子的重用收益发生变化,缓存在磁盘中的数据的重用收益可能会大于内存中的数据,例如算子引用次数增多.因此,当重用磁盘中的数据时,考虑将其迁入内存.此时需要重新计算重用收益,带宽设置为内存带宽,即假设将其存放在内存能够带来的重用收益,以此去跟内存中数据的重用收益相比较,当内存空间充足或者通过替换可以释放足够空间时,将此数据迁入内存.

### 1.3 数据缓存

数据缓存包含 2 个方面的内容,选择重用收益大的数据在查询执行过程中自动缓存以及为数据建立算子索引.

1) 选择数据缓存的依据是收益模型.对于当前执行的查询语句,对其查询计划树中的每个算子的重用收益进行评估,缓存各分支重用收益最大的算子.

2) 为数据建立算子索引.由于数据缓存下来的目的是为了后续重用,因此需要有一种方式能够根据算子找到对应的缓存数据,本文为历史负载中的每个算子生成一个唯一的 ID 标识,在缓存算子的数据时,建立算子 ID 与其数据之间的关联.

### 1.4 数据重用

在改写查询计划时,对于每个算子,先根据其 ID 在缓存空间中寻找数据.若找到,则改写查询计划为从缓存空间中加载数据.本文的数据重用是基于 Partition 粒度的,可能存在一个算子只有部分 Partition 数据在缓存空间的情况.所以在进行数据重用时,使用 bitmap 表示算子的每个 Partition 的数据是否可以重用.若可以重用,则直接读取缓存空间中的数据,否则进行重新计算.

### 1.5 缓存数据管理

本文利用混合介质构建缓存空间,并采用基于收益模型的 Partition 粒度缓存管理策略.当空间充足时,数据优先存放在内存,其次存放在磁盘.当空间不足时,进行替换.数据存储的基本单位是 Partition.因此还需要维护缓存空间 Partition 的信息.数据的重用收益会发生变化,因此对于磁盘上的数据,当其重用收益增大时,考虑将其迁入内存.下面将从写入策略、替换策略、Partition 信息的维护和磁盘数据迁入内存 4 个方面介绍本文的方案.

1) 写入策略.数据写入的基本单位是 Partition.为 Partition 的所有数据申请空间,若内存空间足够,则将其放入内存;当内存空间不足时有 2 种选择,即替换内存中的数据和存放在磁盘.由于本文基于收益模型进行替换,在替换时需要获取缓存空间所有数据的重用收益,替换出重用收益小的数据,替换开销较大.因此本文采用后一种方案,当内存空间不足时将数据存放在磁盘,若磁盘空间仍不足,则表示申请空间失败,需要进行替换.

2) 替换策略.当空间不足时,根据收益模型替换缓存空间中重用收益小的算子结果,优先替换内存中的数据.若不能满足空间需求,则替换磁盘中的数据.内存中被替换出的数据存放在磁盘,磁盘中被替换的数据直接删除.

3) Partition 信息的维护.在替换时根据收益模型替换出重用收益小的 Partition.收益模型依赖于统计信息,因此在缓存空间中需要维护 Partition 的统计信息.另外,还需要建立 Partition 与数据之间的关联,从而保证正确的替换.

4) 磁盘数据迁入内存.当磁盘中数据的重用收益增大时会考虑将其迁入内存.具体方法是从内存申请所需空间,若空间不足,则触发内存替换.通过替换仍不能满足要求时,则放弃迁入内存.如果能够缓存在内存,那么就将磁盘中的数据移动到内存.

## 2 Criss 系统实现

本文基于 Spark SQL 平台实现了第 1 节中的数据重用方案,该系统称为 Criss 系统,如图 3 所示:

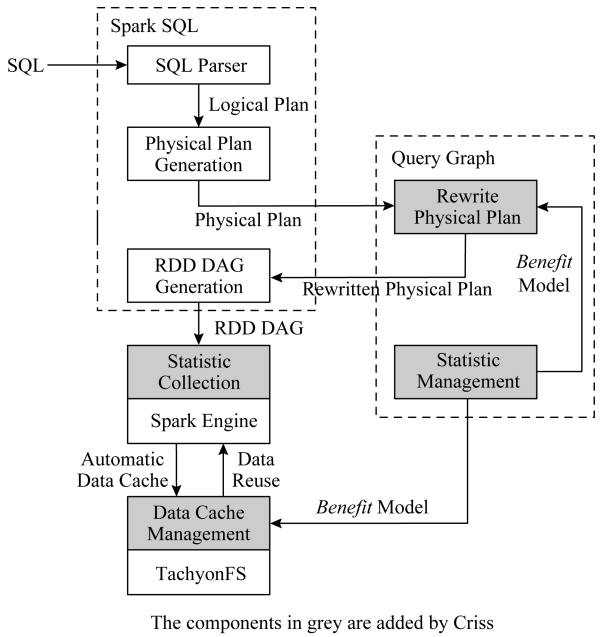



Fig. 3 The implementation of Criss system

图 3 Criss 系统实现图

Criss 系统对 Spark SQL 的修改主要体现在 4 个方面:

1) 增加 Query Graph 组件用于识别重复计算.Query Graph 维护系统的历史负载,负责对翻译后的查询计划进行匹配与改写.由于查询计划的改写依赖于收益模型,在 Query Graph 中还需维护算子的统计信息.

2) Spark SQL 根据改写后的查询计划生成新的 RDD DAG.添加信息收集、数据重用、计算结果自动缓存 3 个功能.

3) 在 Spark 执行引擎中添加统计信息收集功能,统计信息用于收益模型.

4) 利用 TachyonFS[29] 存储重用数据.TachyonFS 是一个基于内存的分布式文件系统,且提供分层存储的功能[30].用户可以配置使用内存、SSD 和磁盘中的一种或者多种介质.为了实现基于收益模型的 Partition 粒度缓存数据管理策略,还需要对 TachyonFS 进行修改.

在 Criss 系统中,SQL 语句的执行流程为:

1) Spark SQL 接收用户输入的 SQL 语句,将其翻译成查询计划.

2) 将查询计划发送给 Query Graph 组件进行匹配,Query Graph 根据历史负载对查询计划进行改写,即标记每个算子是否自动缓存计算结果、是否进行数据重用、是否收集运行时信息.

3) Spark SQL 接收到改写后的物理计划后生成新的 RDD DAG.对于需要自动缓存计算结果的算子,设置其输出 RDD 的 StorageLevel 为 OFF_HEAP,表示将计算结果存在 Tachyon 上;对于可以重用数据的算子,生成新的 RDD 从缓存空间加载数据;对于需要收集运行时信息的算子,在其执行逻辑中插入具有信息收集功能的代码.

4) 将上一步生成的 RDD DAG 提交给执行引擎执行,根据 RDD 的执行逻辑进行统计信息收集、数据重用和计算结果缓存.在缓存数据时,若空间不足会产生替换,替换的依据是收益模型,优先保存重用收益大的数据,因此 TachyonFS 还需从 Query Graph 处获取统计信息.当有需要收集统计信息的算子时,在 Query 执行完后会将统计信息发送给 Query Graph 进行更新.

### 2.1 物理计划改写

本文在 Spark SQL 系统中添加新的组件 Query Graph 用于识别重复计算.识别重复计算需要存储历史查询负载.在 Query Graph 中采用如 1.1 节所述的树结构 QTree 来存储.

除了存储历史查询负载以识别重复计算之外,Query Graph 还需要根据识别的结果对当前查询计划进行改写,包括:1)为匹配成功的算子添加数据重用功能;2)选择此次执行过程中需要自动缓存的计算结果;3)为没有统计信息的算子添加信息收集功能.为此,在当前查询计划的每个算子中添加 3 个布尔类型的属性(*reuse*,*cache*,*collect*),分别表示在执行过程中是否重用数据、是否自动缓存数据、是否收集运行时统计信息.另外,算子还需包含一个全局唯一的 ID,QTree 中的算子亦使用此 ID.当算子可以重用数据时,根据此 ID 即可在缓存空间中找到对应数据.当算子的计算结果需要自动缓存时,此 ID 决定了数据在缓存空间中存放位置(文件名);当需要收集运行时统计信息时,此 ID 保证了在任务执行完后将统计信息保存在 QTree 中相应的节点.

对于算子不完全匹配的情况,本文通过改写查询

计划创造出数据重用机会.对于 Project 算子,当前待匹配算子记为 $A$.若在 QTree 中存在类型为 Project 的算子 $B$,使得算子 $A$ 的 Project 条件为算子 $B$ 的 Project 条件的子集,且算子 $B$ 的数据已在缓存中,则表示算子 $A$ 的结果可以在算子 $B$ 结果的基础上再次进行 Project 操作得到.为此,查询计划改写为在算子 $A$ 与其子算子之间插入新的算子 $B$.

对于 Exchange 算子,其功能是对子算子的数据进行重划分,并将划分后的数据分发给不同的机器进行处理.例如根据连接操作的键(join key)进行划分,那么不同表中具有相同连接操作键的记录会被发送到相同的机器,使得连接操作在这些机器上可以并行执行.在 Spark SQL 中用户可以设置划分的数目,若仅仅因为用户设置数目的不同而导致 Exchange 算子的数据不能够重用,则会丧失很多重用机会.为此本文在 Exchange 算子不匹配的情况下会判断是否是因为划分数目不同引起的.若是,则对查询计划进行改写,在原 Exchange 算子和其子算子之间插入一个新的 Exchange 算子.

### 2.2 信息收集

收益模型中需要 $T_{exec}$ 和 *size* 运行时收集统计信息.然而,收集信息会在一定程度上影响查询语句的执行效率.为此本文只有在算子无统计信息时进行收集.对于需要收集统计信息的算子,在其执行流程中插入信息统计功能代码.

1) 任务内部独立收集统计信息.任务执行时,在迭代过程中获取算子从原始数据产生输出每行数据的时间及输出每行数据的数据量.

2) 汇总统计信息.由于任务在执行完后会将执行结果发送给 Spark 的任务调度器 DAG Scheduler,调度器根据任务的执行结果进行下一步的调度.利用这一点,本文对执行结果的格式进行修改,将任务内部收集的统计信息添加进执行结果中发送给调度器.同时,本文在调度器里增加少量代码,对各个任务的统计信息进行汇总.另外,本文对 Spark SQL 应用程序的 Driver 进行了修改,在查询计划执行完毕后,Driver 从调度器中获取统计信息,并发送给 Query Graph 模块.Query Graph 模块根据 ID 对 QTree 中涉及到的算子的统计信息进行更新.

### 2.3 数据缓存

Query Graph 在进行查询计划匹配时,对于在 QTree 中匹配节点有统计信息的算子,基于收益模型计算其重用收益.对于重用收益大的算子,将其 *cache* 属性为 true,表示在此次执行过程中自动缓存.

对于需要自动缓存的算子,在生成 RDD DAG 片段时,设置其输出 RDD 的 *StorageLevel* 为 OFF_HEAP,表示在执行过程中将数据缓存在 Tachyon 中.

原始 Spark 的 OFF_HEAP 方式只支持应用程序内同一 RDD 对象的数据重用.为了支持不同 RDD 对象数据的重用,本文扩展了 Spark 的 OFF_HEAP 缓存机制.对于在 Spark SQL 场景下需要自动缓存数据的 RDD,更改其输出 RDD 的 *name* 属性为 *operator_operatorId_splitIndex*,其中 *operatorId* 为算子在 QTree 中算子的 ID,具有全局唯一性,因而可以用于不同 RDD 对象之间的数据重用.

本文将缓存数据存放在 TachyonFS 中,每个 Partition 的数据存为一个文件.缓存空间的数据组织如图 4 所示,采用有 3 级目录结构.全局目录下存放算子目录(*operatorId*),算子目录下存放 Partition 数据文件.

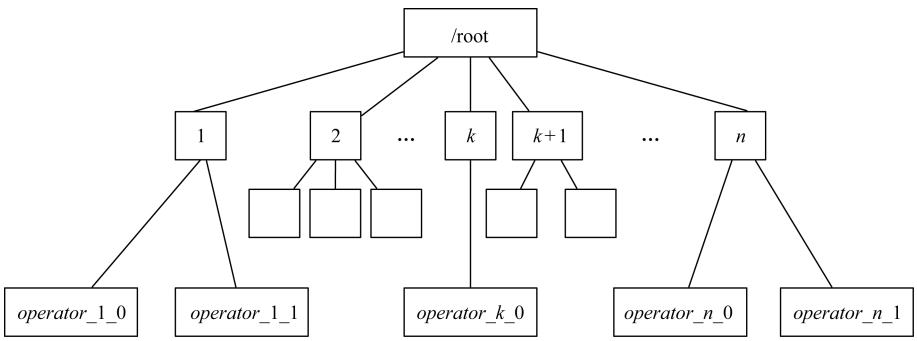



Fig. 4 Organization of cached data

图 4 缓存空间数据组织方式

### 2.4 缓存数据管理

本文在 Tachyon 中实现基于收益模型的 Partition 粒度的管理策略.为此,本文对 Tachyon 进行扩展,主要包括:

1) 写入数据时以 Partition 为粒度.

2) 空间不足发生替换时以 Partition 为粒度,且替换依据为收益模型.

3) 向外提供将磁盘上的 Partition 数据迁入内存的接口.当引用次数增多导致磁盘数据的重用收益增大时,由上层 Spark 系统调用将其迁入到内存,提高数据传输速度,进一步增加重用收益.

## 3 性能评测

为了评价数据重用机制在 Spark SQL 中的性能,本文采用 TPC-H Benchmark[31] 对 Criss 系统与原始 Spark SQL 系统的查询性能进行评测,并对本文所提出的关键技术进行评测,包括混合介质、收益模型和 Partition 粒度重用.通过与现有技术的对比,表明了本文所提出的方法更适合 Spark SQL 查询分析.

### 3.1 测试负载及测试平台

本文使用 TPC-H 来评测 Criss 系统的性能.测试中数据总量为 100 GB.为了模拟实际场景中的重复计算,我们采用 TPC-H 的 Query Stream 的执行模式,顺序执行 100 条查询语句.由于实际应用中,有些重复计算来自于多次执行相同的查询语句,另一些重复计算则来自于多条查询之间有交集(即查询计划树的一部分是相同的),例如多条查询都对同一张表做相同的筛选(Filter)或计算(Sort 或 Aggregation).为了评测不同的重复计算场景,我们用 TPC-H 的 22 条查询语句构造出 100 条查询语句,分别模拟以下不同的场景:

1) Random-QS.该负载模拟局部性差的场景,即重复执行的查询语句呈随机分布.从 TPC-H 的 22 条查询语句中随机选择共 100 条语句.有的语句会重复多次,但不同语句的重复次数不相同,何时重复执行也不相同.

2) Zipf-QS.该负载模拟局部性强的场景,即大部分操作都集中在少数的热点查询语句上.从 TPC-H 的 22 条查询语句中按照 Zipf 分布选择共 100 条语句.

3) Random-QS-v.该负载模拟查询语句之间有交集的情形.对 TPC-H 中的每一条查询语句,通过改变参数,产生与原始查询语句匹配率约为 60%和 30%的 2 条语句,最终有 66 条查询语句.然后,从这 66 条语句中随机选择共 100 条语句,并控制每条语句的出现不超过 2 次,以消除查询语句完全匹配的影响.

本文的测试平台为 4 台物理机器搭建的 Spark 和 Tachyon 集群.每台机器有 12 个物理核,32 GB 内存,3 块 1 TB 的数据盘,运行在 64 位 CentOS 系统上,内核版本为 2.6.32,Java 版本为 1.7.0.集群中 1 台机器作为 Spark 和 Tachyon 的 master 节点,其余机器作为 worker 节点.原始数据存储在 HDFS 中,对应的 Hadoop 版本为 2.2.0,块大小设置为 256 MB,副本数为 3.Spark 的版本为 1.5.1,集群中 1 个节点上同时运行的任务数最多为 16,每个节点内存设置为 30 GB,Shuffle 时 Partition 总数目为 200.Tachyon 的版本为 0.7.1,Block 大小为 128 MB.

### 3.2 重用收益评测

本小节对 Criss 系统的性能进行评测,使用如 3.1 节所述的 3 种负载,3 种负载下算子的重复率分别为 81%,85%,62%.缓存空间总容量设置为 300 GB.在 Random-QS 和 Zipf-QS 负载下,内存空间容量设置为 3 GB.而在 Random-QS-v 负载下,算子的重复率较低,表明负载中对算子的访问越分散,那么,由收益模型选择进行自动缓存的数据就越多,因此内存容量设置为 18 GB.

图 5 给出了 Criss 和 Spark SQL 在 3 种负载下的总执行时间,Criss 系统分别可以带来约 46%,68%,58%的性能提升.

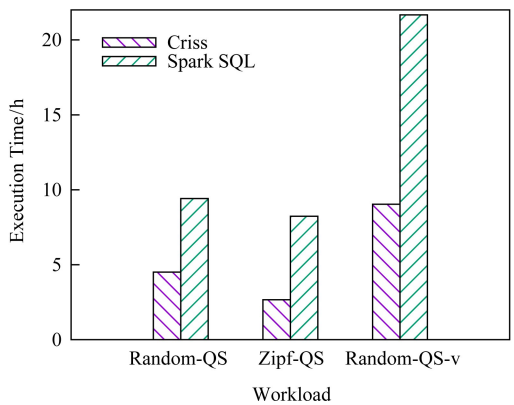



Fig. 5 Reuse benefit evaluation

图 5 重用收益评测

### 3.3 混合介质评测

对于混合介质策略,为了表明混合介质相比于单一介质更适合大数据场景,本文扩展了 Criss 系统,使其也可以使用内存或者磁盘单一介质构建缓存空间,然后评测了 Criss 系统在 3 种配置下的性能表现:Criss-Hybrid,Criss-Disk,Criss-Mem,分别对应于混合介质、单一磁盘介质和单一内存介质.

如图 6 所示,混合介质存储相比于单一磁盘能够

提升 7%～13%的性能，相比于单一内存能够提升 10%～27%的性能.

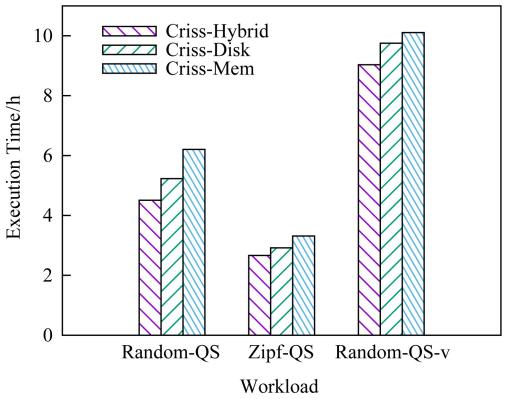



Fig. 6 Comparison of storage medium

图 6 存储介质对比

在 Criss 系统中，会自动缓存重用收益大的算子的计算结果，重用这些算子的结果能减少重复计算，提升系统的性能.然而算子结果的缓存会带来额外的开销，体现在 2 个方面：1)将算子结果写入缓存介质上的存储开销.2)每缓存一个算子的结果就会减慢系统的执行，这是因为 Spark SQL 的执行是 Pipeline 方式.每获取一行数据，就对其进行操作，而每缓存一个算子的结果意味着需要打破这种 Pipeline 执行方式，先获取算子的结果存入缓存介质，然后再继续后面的执行任务，因此会减慢系统的执行.

图 7～9 分别显示了 3 种负载在不同介质下缓存空间的存储和重用情况，分别从算子和数据量的角度进行了对比.由于 Random-QS-v 负载下缓存空间容量不足，会出现算子的部分 Partition 数据重用的情况，因此在此负载下仅对比了数据量.

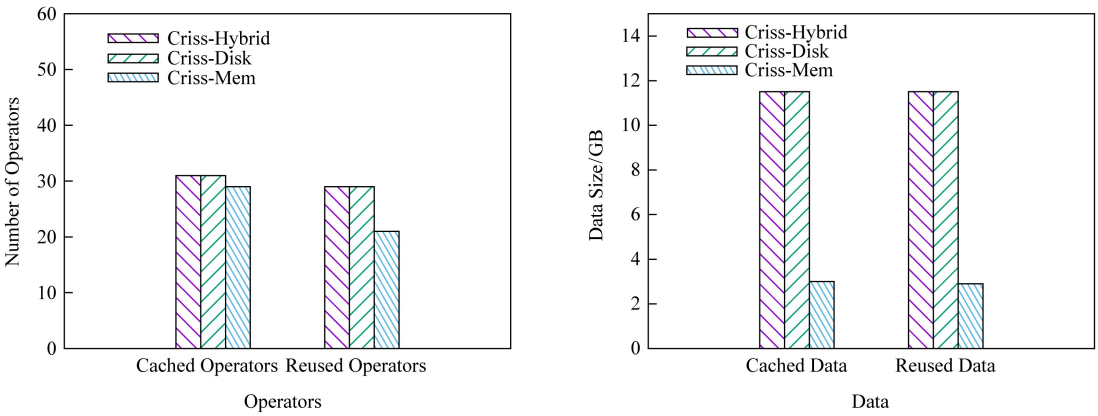



Fig. 7 Cached and reused statistics of Random-QS

图 7 Random-QS 缓存空间存储和重用情况

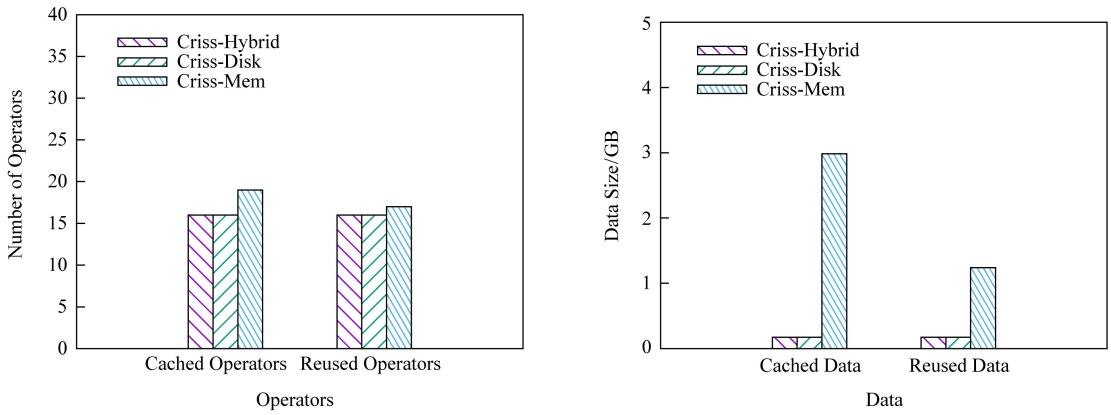



Fig. 8 Cached and reused statistics of Zipf-QS

图 8 Zipf-QS 缓存空间存储和重用情况

从图 7～9 中可以看出：

1) 3 种负载下，磁盘与混合介质的表现都一致，但由于混合介质把一部分数据存放在内存，能够节省这部分数据存入时的写开销以及重用时的读开销，

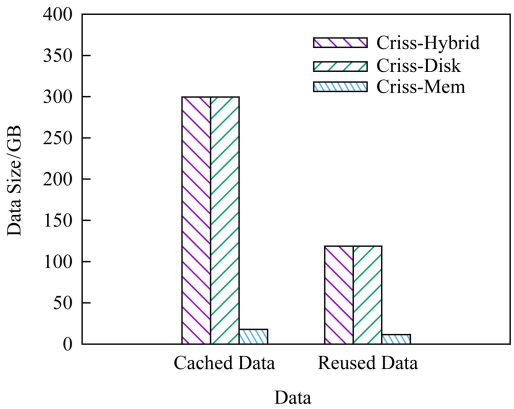



Fig. 9 Cached and reused statistics of Random-QS-v

图 9 Random-QS-v 缓存空间存储和重用情况

因此系统整体性能高于磁盘.在 3 种负载下,混合介质方案写入内存中的数据的比例和从内存中重用数据的比例如表 1 所示:

**Table 1 Data Cached Ratio and Reused Ratio for Hybrid Storage**

**表 1 混合介质方案下数据在内存中的存储及重用比例 %**

| Workload | Percentage of Cached Data | Percentage of Reused Data |
|---|---|---|
| Random-QS | 26 | 26 |
| Zipf-QS | 100 | 100 |
| Random-QS-v | 6 | 5 |

2) 在 Random-QS 和 Random-QS-v 负载下,内存比混合介质缓存和重用了更少的算子结果.这 2 种负载下,使用单一内存介质时,缓存空间的容量成为瓶颈.从理论上来讲,内存缓存了更少的算子结果,节省了缓存开销,但是重用更少的算子意味着更少的重用收益.

在 Random-QS 负载下,本文选取了在混合介质和内存配置下缓存情况不同而重用情况相同的 Query 集合以比较缓存开销,选取了在 2 种配置下缓存情况相同而重用情况不同的 Query 集合以比较重用收益.实验结果显示,混合介质相比于内存额外重用算子结果所带来的性能收益和额外缓存算子结果带来的缓存开销分别为 84 min 和 5 min,额外缓存的算子结果带来的重用收益大于其缓存开销.而内存受限于容量的大小不能缓存更多有价值的数据,因此性能低于混合介质.

在 Random-QS-v 负载下,混合介质配置下由收益模型选择需要缓存的数据总量已超出了缓存空间的容量,即需要缓存的数据总量至少为 300 GB.而使用单一内存介质时,只能缓存 18 GB 的数据,占需缓存数据总量的比例不到 6%,在存储过程中发生了频繁的替换,导致其存储效率不高.本文选取重用情况相同而缓存情况不同的 Query 集合进行统计,发现在混合介质配置下这些 Query 的执行时间总和为 1.7 h,而内存配置下执行时间总和为 2.1 h,内存的缓存开销大于混合介质,而内存又重用了更少的数据,因此最终的性能不如混合介质.

3) 在 Zipf-QS 负载下,内存比混合介质缓存和重用了更多的算子结果.这是因为,根据 1.2 节的收益模型,缓存数据的开销为 $T_{\text{store}}=size/bw$.在存储介质为内存时,由于内存带宽大,因此会缓存大量的数据到内存.而对于混合介质,由于 $bw$ 的计算是按照混合介质带宽的最坏情况,即磁盘带宽进行计算的,因此在混合介质中缓存的数据反而更少.而且,内存额外缓存的 3 个算子结果中,只有一个后续被重用了,且重用时带来的性能收益小于 1 s,抵消不了其额外缓存数据的开销,因此整体性能不如混合介质.

综上所述,使用磁盘单一介质时,缓存空间存储与重用的情况与混合介质一致,但由于其加载速度慢,缓存数据带来的重用收益比混合介质小,因此系统的整体性能不如混合介质;使用内存单一介质时,在缓存空间容量充足的情况下,缓存开销较大,在缓存空间容量不足的情况下,重用收益较小,因此系统的整体性能也不如混合介质.

### 3.4 收益模型评测

在选择需要缓存的数据时,本文使用如 1.2 节所述的收益模型.与已有研究工作的收益模型不同的是,本文考虑了缓存数据读写时间的影响,能更为准确地评估重用收益.为了评测收益模型,本文将 Recycler[9] 的收益模型实现在 Criss 系统中.Recycler 中使用 $ref\times cost/size$ 表示数据的重用收益,其中 $ref$ 表示算子的引用率,$cost$ 表示执行时间,$size$ 表示数据量的大小.本节对本文的收益模型 Criss-Benefit 与 Recycler 中的收益模型 Criss-Recycling 进行了性能对比评测.

图 10 显示了负载在不同收益模型下的执行时间,从图 10 中可以看出,本文的收益模型 Criss-Benefit 相比于 Recycler 中的收益模型 Criss-Recycling 能够提升 10%~25%的性能.

为了进一步观察本文收益模型的优势所在,图 11~13 显示了缓存空间数据缓存与重用的情况,与 3.3 节类似,对于 Random-QS 和 Zipf-QS 从算子

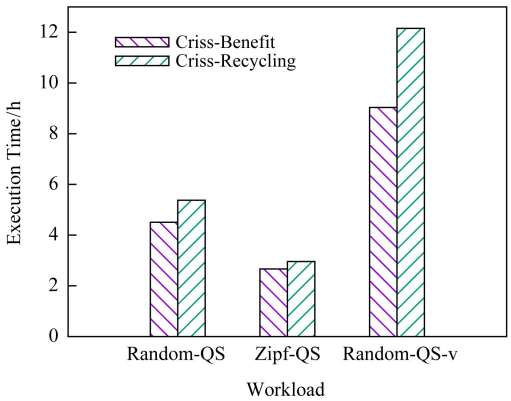



Fig. 10 Comparison of benefit model

图 10 收益模型对比

角度和数据量 2 个角度进行对比,而对于 Random-QS-v,仅从数据量角度进行对比.测试结果表明:

1) 在 Random-QS 和 Zipf-QS 负载下,Criss-Recycling 均比 Criss-Benefit 缓存和重用了更多的算子结果.Criss-Recycling 缓存了更多的算子结果,有更多的缓存开销,然而,重用了更多的算子结果会有更多的重用收益.

本文选取了在 2 种收益模型配置下缓存情况不同而重用情况相同的 Query 集合以比较缓存开销,选取了在 2 种配置下缓存情况相同而重用情况不同的 Query 集合以比较重用收益.2 种负载下,Criss-Recycling 相比于 Criss-Benefit 额外重用算子结果所带来的性能收益以及额外缓存算子结果带来的缓存开销如表 2 所示.从表 2 可以看出 Criss-Recycling 额外缓存的算子结果带来的缓存开销大于性能收益,其收益模型缓存了很多重用价值低的数据,因此整体性能不如 Criss-Benefit.

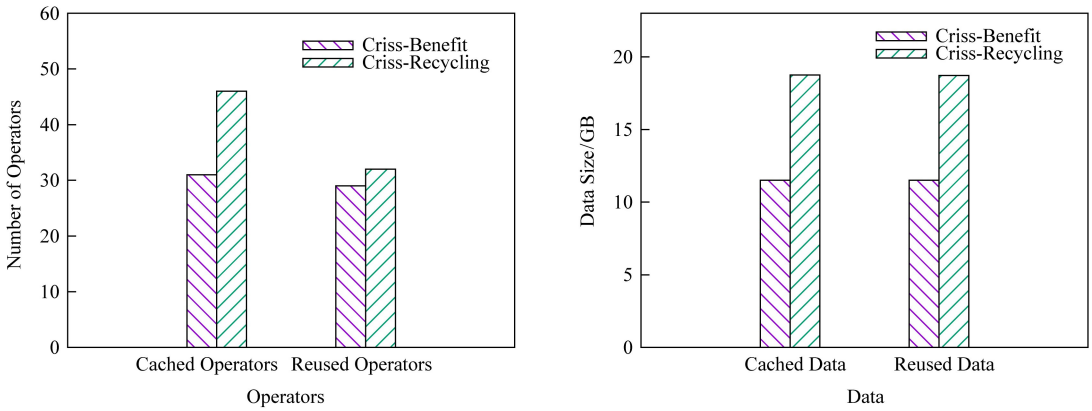



Fig. 11 Cached and reused statistics of Random-QS

图 11 Random-QS 缓存空间存储和重用情况

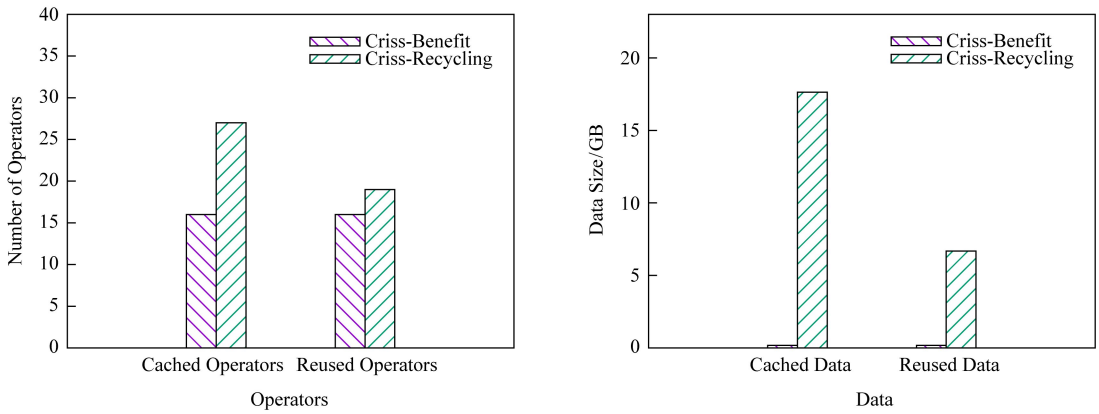



Fig. 12 Cached and reused statistics of Zipf-QS

图 12 Zipf-QS 缓存空间存储和重用情况

本文详细分析了 Random-QS 负载下 Criss-Recycling 额外缓存的算子结果,发现其额外缓存数据的 14 个算子中,有 13 个在使用本文的收益模型时计算出的重用收益小于 0,即计算开销小于缓存开销,而 Criss-Benefit 只缓存重用收益大于 0 的数据,相比之下,Criss-Recycling 没有这个限制,因此

Criss-Recycling 额外缓存了重用价值低的数据,这也说明了本文的收益模型能够更为准确地选择重用价值大的数据.

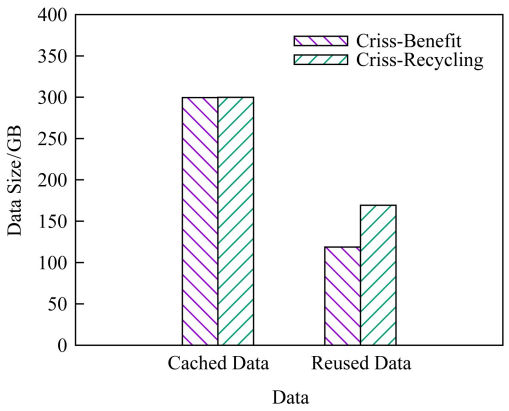



Fig. 13 Cached and reused statistics of Random-QS-v

图 13 Random-QS-v 缓存空间存储和重用情况

**Table 2 Extra Performance Gains and Cache Cost of Criss-Recycling**

**表 2 Criss-Recycling 额外的性能收益及缓存开销** min

| Workload | Performance Gain due to Reusing Data | Performance Cost due to Caching Data |
|---|---|---|
| Random-QS | 9 | 30 |
| Zipf-QS | 14 | 30 |

2) 在 Random-QS-v 负载下,Criss-Benefit 与 Criss-Recycling 缓存的数据一致,但重用的算子结果比 Criss-Recycling 少.二者缓存的数据总量均达到了缓存空间的容量,缓存空间容量受限,根据收益模型选择的算子结果不能够全部存放,只能保留重用收益大的数据,不同收益模型对重用收益的评估标准不同,因此最后保留的数据也不一样.

本文选取缓存情况相同而重用情况不同的 Query 集合进行统计以比较重用收益,发现在 Criss-Recycling 配置下这些 Query 的执行时间总和为 4.6 h,而在 Criss-Benefit 配置下执行时间总和为 3 h,Criss-Benefit 的性能收益大于 Criss-Recyling,说明 Criss-Benefit 在缓存空间中所保留数据的重用收益较大,显示了本文收益模型的有效性.

### 3.5 重用粒度的评测

与算子粒度重用相比,Partition 粒度重用能够充分地利用缓存空间,提高存储效率.为了对这 2 种重用粒度进行性能对比评测,本文在 Criss 系统中也实现了算子粒度重用(Criss-Operator).

图 14 显示了 3 种负载下本文的 Partition 粒度重用和算子粒度重用的性能,测试结果表明:

1) 在 Random-QS 和 Zipf-QS 负载下,Partition 粒度重用只比算子粒度重用的性能稍好一些.这是因为,在空间充足的情况下,算子粒度重用只比 Partition 粒度重用多一些管理开销,即需要等待算子的所有 Partition 都缓存下来,而 Partition 粒度重用则不需要.

2) 在 Random-QS-v 负载下,Partition 粒度重用比算子粒度重用性能高 49%.此时根据收益模型选择的数据不能够全部存放,算子粒度重用会引起频繁的替换,导致缓存效率低及缓存空间利用率不高的问题.在此负载下,算子粒度比 Partition 粒度替换更多的数据,分别为 76 GB 和 45 GB,因此存储效率低.另外,本文观察了 Partition 粒度下重用的 118 GB 数据,发现有 25 GB 的数据来自于算子的部分 Partition 被重用,算子的少量 Partition 被替换了出去.而在算子粒度下这些算子的所有 Partition 都会被替换出去,从而不能受益于大部分 Partition 的重用.本文选取缓存情况相同而重用情况不同的查询语句进行统计,发现在 Partition 粒度下这些语句的执行时间总和为 1.7 h,而在算子粒度下执行时间总和为 4 h,说明算子粒度下所保留数据的整体重用收益较低.

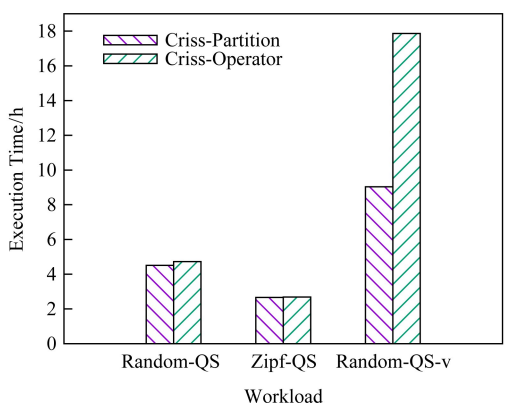



Fig. 14 Comparison of reuse grain

图 14 重用粒度对比

## 4 相关工作

### 4.1 重复计算的识别

识别重复计算是进行数据重用的前提,目前的研究工作主要是通过查询匹配与改写完成,查询匹配用于识别完全相同的计算,查询改写能够增加重用机会.查询匹配目前主要有三大类方式:SQL 字符串匹配、规范化查询模板、基于算子的匹配.

1) SQL 字符串匹配[17,27].SQL 语句表达为符合一定语法的字符串,SQL 字符串匹配是对多条 SQL 语句进行字符串匹配,如果它们的字符串相同,则说明它们是相同的计算.因此,只需要执行其中一条语句,它的执行结果就可以被其他语句重用.Shang[17]通过计算 SQL 字符串的 Hash 值来加速匹配,即仅当 Hash 值匹配时才需对字符串进行匹配.这种方式虽然提高了匹配速度,但是,SQL 字符串匹配只是查询匹配的充分条件,即使 2 条 SQL 语句的字符串不相同,它们也可能执行相同的计算.

2) 规范化查询模板[11,13,15].这种方法是定义一个查询模板,并按模板重写每条查询,即根据具体的查询来填充模板中的各个元素.通过对重写后的查询语句进行匹配来识别重复计算.但是,与 SQL 字符串匹配类似,规范化查询模板也是基于整条查询语句的匹配.它们都只能重用查询的最终结果,无法识别查询语句间有交集的情形,无法重用查询的中间结果.并且,不是所有的查询都能用规范化模板来表示,因此,其应用场景有限.例如,广泛使用的连接查询有多种类型,包括内连接(inner join)、左外连接(left outer join)、完全外连接(full outer join)等.不同类型的连接查询产生不同的计算结果,而 Hawc[11]中的规范化模板并不能够区分不同类型的连接查询.

3) 基于算子的匹配[7,9,13,16,18,20,28].数据处理系统在执行 SQL 查询时,会将其翻译成查询计划.查询计划是由算子组成的树结构,表示数据的处理流程.基于算子的匹配则是自底向上逐个匹配查询计划树中的各个算子.算子匹配的条件是算子类型相同且表达式相同.算子匹配成功说明是相同的计算,即重复计算.与前面 2 种方法不同,基于算子的匹配能够识别查询语句间有交集的情形(称为部分匹配),即查询计划树中的一部分是相同的计算.因此,该方法能够重用查询的中间结果,而不仅仅是整条查询的最终结果.而且,查询计划树表达了数据处理流程,即使 2 条查询的 SQL 字符串不同,如果它们执行相同的计算,它们的查询计划树就是相同.因此,基于算子的匹配能够识别出更多的重复计算.

鉴于基于算子的匹配有上面 3 个优势,本文的 Criss 系统采用该方法.

### 4.2 数据缓存位置

由于传输速度(即带宽)不同,数据缓存在不同存储介质上带来的重用收益也不同.内存的带宽高,但容量较小,能够容纳的缓存数据量较少,多用于单机场景[9-10,12,27].相比于内存,磁盘虽然带宽低,但其容量很大,能够缓存更多的数据,因此更适合数据规模较大的场景,在传统数据库[11,13]和大数据平台上[14-15,17]都有应用.在大数据平台上,大多利用 HDFS 来存储缓存数据.利用 HDFS 提供的文件接口,缓存和管理数据非常简便.HDFS 为了容错采用了多副本机制.但是,在数据重用场景下数据丢失了可以重新计算,不必存储多个副本.多副本机制对于数据重用场景来说反而导致了磁盘空间的浪费.

现有的数据重用只采用单一的存储介质,而对于缓存重用数据,内存和磁盘各有其优劣.本文的 Criss 系统采用混合存储的方案,扬长避短,充分发挥各存储介质的优势,以提高系统的整体重用收益.具体来讲,将重用收益大的数据存放在内存,重用收益较小的数据存放在磁盘.在受益于内存的高速数据传输能力的同时,利用磁盘缓存更多的数据,最大化缓存空间的重用收益.

### 4.3 缓存数据的选择

缓存数据选择方法主要有两大类:基于规则的选择和基于收益模型的选择.

1) 基于规则的选择.系统制定出一定的规则,在查询执行的过程中符合规则的计算结果都会被缓存下来[7,13-14,16,18].ReStore[14]根据算子类型选择缓存数据,即选择那些计算开销大、输入数据量大而结果数据量小的算子(例如 Filter 和 Join),将它们的计算结果进行缓存.但是,根据表达式的不同,算子会有多种形式,而这种方法对于同类算子没有区分度.例如,Filter 算子在筛选条件很紧时,结果数据量才很小,重用收益才大.DynaMat[13]提出了 3 种规则,分别根据 LRU,LFU 和计算结果大小对缓存空间中的数据进行管理.SCOPE[7],SQL Server[16],SparkCruise[18]使用简单的启发函数(例如 Top $K$)进行缓存数据选择.这些方案考虑的因素都比较单一,往往不能选择出最有价值的数据进行缓存.

2) 基于收益模型的选择.根据多种因素,建立收益模型来评估计算结果的重用收益,根据重用收益大小进行缓存数据的管理,以提高缓存空间的整体重用收益.目前的研究工作中所考虑的因素有计算开销[9-13,15,27]、算子类型[11]、数据量[9,12-13,27]、更新率[12]、引用次数[9,11,13]或者引用率[12,27]、最近访问时间[11]等.例如,Recycler[8]中的收益模型为 $ref \times cost/size$,其中 $ref$ 表示计算结果的引用次数,$cost$ 表示计算结果的执行时间,$size$ 表示计算结果的数据量.

相比于基于规则的选择方法,基于收益模型的选择方法能够更为准确地评估数据的重用收益.但是,现有的收益模型都没有考虑数据读写时间的影响.数据的每一次重用都需要从缓存介质上读取数据到应用程序内存,且在重用之前还需要将数据写入缓存介质上.在数据规模较大时,数据读写时间对重用性能的影响也较大.而且,数据读写时间,不仅与数据量有关,还受存储介质传输速度的影响.在不同的存储介质上数据读写时间是不相同的.现有的收益模型没有考虑采用混合存储介质,也没考虑不同介质下的不同数据读写时间.本文提出了的重用收益模型,不仅考虑了数据读写时间的影响,而且针对混合介质,考虑了不同介质上数据读写时间的差异.因此,本文提出的收益模型更为精准.

### 4.4 数据重用粒度

在传统数据库中,多采用算子粒度的数据重用[7,9-13,16,27].查询语句的查询计划树中各算子产生的计算结果不可细分,只能作为一个整体存储与重用.在进行数据替换时,也是以算子为单位进行,即算子的结果作为整体替换,而不能只替换算子结果的一部分.后来出现的分布式大数据处理平台在应用数据重用技术时,也沿用了基于算子粒度的方式[14-15,17-18,20].

在大数据场景下,基于算子粒度的重用方式逐渐凸显出局限性.一方面,分布式大数据处理平台采用并行处理,算子的计算结果是分布于集群中的多台机器上,由很多 Partition 组成;另一方面,基于算子粒度的缓存管理不能充分利用所有机器上的缓存空间,降低系统的整体效率.因为集群中各个机器运行的任务并不一样,它们处理的数据也不一样,随着不断地执行各种任务,各个机器上缓存空间的使用率通常是不相同的,有的机器缓存空间满了,其他机器的缓存空间还充足.如果基于算子粒度的缓存就不能充分利用各个机器的缓存空间.而本文提出 Partition 粒度的数据重用,采用更细粒度的缓存替换,能够更充分缓存空间,提到系统的整体效率.

### 4.5 其他重用方法

HashStash[19]提供了一种特别的重用角度,与其他相关工作和本文的方法都不同,它将重用的机会锁定在算子内部数据结构——Hash 表上,通过分析查询计划发现缓存 Hash 表的机会,并在查询优化阶段对查询计划进行改写,以重用缓存的 Hash 表.在选择缓存的 Hash 表时,它也使用了基于收益模型的评估方法.

## 5 结束语

针对分布式大数据处理平台,以减少重复计算为目标,本文提出了基于收益模型的数据重用机制:1)采用基于算子的匹配与改写方法识别重复计算;2)针对混合介质,提出一种新的收益模型,更为准确地评估数据的重用收益;3)利用数据集分布存储的特性,提出了 Partition 粒度的数据重用,以提升数据的存储效率和缓存空间的利用率.我们基于 Spark SQL 平台实现了具有数据重用功能的 Criss 系统.实验结果表明,本文提出的数据重用技术显著提升了查询性能,与 Spark SQL 相比,查询性能提升了 46%~68%.

## 参 考 文 献


[1] Apache Spark. Spark [OL]. [2019-10-23]. http://spark.apache.org/

[2] Zaharia M, Chowdhury M, Franklin J, et al. Spark: Cluster computing with working sets [C/OL] //Proc of USENIX HotCloud'10. Berkeley, CA: USENIX Association, 2010 [2019-10-23]. https://www.usenix.org/legacy/events/hotcloud10/tech/full_papers/Zaharia.pdf

[3] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing [C] //Proc of USENIX NSDI'12. Berkeley, CA: USENIX Association, 2012: 15-28

[4] Apache Spark. Apache Spark FQA [OL]. [2019-10-23]. https://spark.apache.org/faq.html

[5] Armbrust M, Xin R, Lian Cheng, et al. Spark SQL: Relational data processing in Spark [C] //Proc of ACM SIGMOD'15. New York: ACM, 2015: 1383-1394

[6] He Bingsheng, Yang Mao, Guo Zhenyu, et al. Wave computing in the cloud [C/OL] //Proc of USENIX HotOS'09. Berkeley, CA: USENIX Association, 2009 [2019-10-23]. https://www.usenix.org/legacy/event/hotos09/tech/full_papers/he/he.pdf

[7] Jindal A, Qiao Shi, Patel H, et al. Computation reuse in analytics job service at Microsoft [C] //Proc of ACM SIGMOD'18. New York: ACM, 2018: 191-203

[8] Agrawal S, Chaudhuri S, Narasayya V. Automated selection of materialized views and indexes in SQL databases [C] //Proc of VLDB'00. San Francisco: Morgan Kaufmann, 2000: 496-505

[9] Nagel F, Boncz P, Viglas S. Recycling in pipelined query evaluation [C] //Proc of ICDE'13. Piscataway, NJ: IEEE, 2013: 338-349

[10] Ivanova M, Kersten M, Nes N, et al. An architecture for recycling intermediates in a column-store [C] //Proc of ACM SIGMOD'09. New York: ACM, 2009: 309-320

[11] Perez L, Jermaine C. History-aware query optimization with materialized intermediate views [C] //Proc of ICDE'14. Piscataway, NJ: IEEE, 2014: 520-531

[12] Shim J, Scheuermann P, Vingralek R. Dynamic caching of query results for decision support systems [C] //Proc of the 11th Int Conf on Scientific and Statistical Database Management. Piscataway, NJ: IEEE, 1999: 254-263

[13] Kotidis Y, Roussopoulos N. DynaMat: A dynamic view management system for data warehouses [C] //Proc of ACM SIGMOD'99. New York: ACM, 1999: 371-382

[14] Elghandour I, Aboulnaga A. ReStore: Reusing results of MapReduce jobs [J]. Proceedings of the VLDB Endowment, 2012, 5(6): 586-597

[15] Xie Heng, Wang Mei, Le Jiajin. A data reusing strategy based on Hive [C] //Proc of Int Conf on Data Science and Advanced Analytics. Piscataway, NJ: IEEE, 2014: 367-373

[16] Goldstein J, Larson P. Optimizing queries using materialized views: A practical, scalable solution [C] //Proc of ACM SIGMOD'01. New York: ACM, 2001:331-342

[17] Shang Hui. Reusing results in big data frameworks [D]. Stockholm, Sweden: KTH Information and Communication Technology, 2013

[18] Roy A, Jindal A, Patel H, et al. SparkCruise: Handsfree computation reuse in Spark [J]. Proceedings of the VLDB Endowment, 2019, 12(12): 1850-1853

[19] Dursun K, Binnig C, Cetintemel U, et al. Revisiting reuse in main memory database systems [C] //Proc of ACM SIGMOD'17. New York: ACM, 2017: 1275-1289

[20] Camacho-Rodriguez J, Colazzo D, Herschel M, et al. Reuse-based optimization for Pig Latin [C] //Proc of ACM CIKM'16. New York: ACM, 2016: 2215-2220

[21] Nykiel T, Potamias M, Mishra C, et al. MRShare: Sharing across multiple queries in MapReduce [J]. Proceedings of the VLDB Endowment, 2010, 3(1/2): 494-505

[22] Psaroudakis I, Athanassoulis M, Ailamaki A. Sharing data and work across concurrent analytical queries [J]. Proceedings of the VLDB Endowment, 2013, 6(9): 637-648

[23] Bello R G, Dias K, Downing A, et al. Materialized views in Oracle [C] //Proc of VLDB'98. San Francisco: Morgan Kaufmann, 1998: 659-664

[24] Chaudhuri S, Krishnamurthy R, Potamianos S, et al. Optimizing queries with materialized views [C] //Proc of ICDE'95. Piscataway, NJ: IEEE, 1995: 190-200

[25] Gupta A, Mumick I S. Maintenance of materialized views: Problems, techniques, and applications [J]. IEEE Data Engineering Bulletin, 1995, 18(2): 3-18

[26] Gunda P, Ravindranath L, Thekkath C, et al. Nectar: Automatic management of data and computation in datacenters [C] //Proc of OSDI'10. Berkeley, CA: USENIX Association, 2010: 75-88

[27] Scheuermann P, Shim J, Vingralek R. WATCHMAN: A data warehouse intelligent cache manager [C] //Proc of VLDB'96. San Francisco: Morgan Kaufmann, 1996: 51-62

[28] Finkelstein S. Common expression analysis in database applications [C] //Proc of ACM SIGMOD'82. New York: ACM, 1982: 235-245

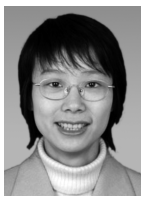
[29] Li Haoyuan, Ghodsi A, Zaharia M, et al. Tachyon: reliable, memory speed storage for cluster computing frameworks [C/OL] //Proc of ACM Socc'14. New York: ACM, 2014 [2019-10-23]. https://dl.acm.org/doi/pdf/10.1145/2670979.2670985?download=true

[30] Alluxio Inc. Alluxio storage management [OL]. [2019-10-23]. https://docs.alluxio.io/os/user/stable/en/advanced/Alluxio-Storage-Management.html

[31] TPC. TPC-H [OL]. [2019-10-23]. http://www.tpc.org/tpch/

**Shen Yijie**, born in 1989. PhD candidate. His main research interests include big data analytics and parallel computing.

**Zeng Dan**, born in 1991. Master. Her main research interests include big data management and parallel computing.



**Xiong Jin**, born in 1968. PhD, professor, PhD supervisor. Senior member of CCF. Her main research interests include storage systems, file systems and big data storage and management.