密级：_____

浙江大学

硕 士 学 位 论 文

论文题目 基于副本选择的大数据实时查询处理并行调度

作者姓名 赵宇亮

指导教师 陈 岭 副教授

陈根才 教授

学科(专业) 计算机应用技术

所在学院 计算机科学与技术学院

提交日期 2015.01.23

A Dissertation Submitted to Zhejiang University for the Degree of Master of Engineering

TITLE: Replica Selection Based Scheduling for Big Data Real-Time Query Processing

Author: Yuliang Zhao

Supervisor: Associate Professor Ling Chen

Professor Gencai Chen

Subject: Computer Application Technology

College: Computer Science and Technology

Submitted Date: 2015.01.23

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得 浙江大学 或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名：　　签字日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解 浙江大学 有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权 浙江大学 可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后适用本授权书）

学位论文作者签名：　　导师签名：

签字日期：　　年　　月　　日　　签字日期：　　年　　月　　日

摘要

查询处理并行调度是分布式查询优化的关键步骤。针对目前调度方法不适用于采用 Massively Parallel Processing (MPP)计算框架的大数据实时查询系统（如 Impala）的问题，本文提出基于副本选择的大数据实时查询处理并行调度方法。该方法将所有查询分为单表查询和多表查询两类：若是单表查询，则首先根据数据分布构造流网络，然后使用 SRPushRelabelBinary 算法选择副本，最后选择执行节点；若是多表查询，则根据本文提出的代价模型搜索近似最优的调度策略。本文定义查询处理的代价为从查询开始处理时刻到预估所有连接操作完成时刻的时间间隔。该代价模型综合考虑通信的代价、并行执行和集群的负载。本文利用 Maxdiff(V, A)直方图估计中间结果，以提高代价模型的准确度。将本文提出的查询处理并行调度方法集成到 Impala2.0 系统，并在 TPC-DS 数据集上进行了实验，结果表明，集成后的 Impala 系统的查询响应时间比集成前的减少了 10%~30%。

关键词：副本选择，并行处理，调度，代价模型，大数据实时查询

Abstract

Scheduling for parallel query processing is a critical phase in distributed query optimization. Current scheduling method cannot be applied to big data real-time query system which is a Massively Parallel Processing (MPP) database engine, such as Impala. To solve this problem, we proposed a replica selection based scheduling method. The method firstly put all queries into two categories: retrieving data from a single table and retrieving data from multiple tables. If a query retrieved data from a single table, the method constructed a flow network according to data distribution, SRPushRelabelBinary algorithm was used to select replica, and then selected execution node. If a query retrieved data from multiple tables, the method tried to find near optimal scheduling strategy with a cost model. The cost of query processing was defined as the interval between the starting time of query execution and the estimated time when all join operators finish. The cost model considered communication cost，parallel execution and the load of the cluster. Maxdiff(V, A) histogram was used to estimate intermediate result and improve the accuracy of the cost model. The proposed method was implemented in Impala2.0, experiments on queries from TPC-DS benchmark indicated that the method can reduce response time by 10%~30%.

Keywords：replica selection, parallel processing, schedule, cost model, big data real-time query

目录

摘要 i
Abstract ii
第 1 章 绪论 1
1.1 研究背景 1
1.2 课题研究国内外现状 4
1.2.1 任务模型 4
1.2.2 代价模型 4
1.2.3 调度方法 7
1.2.4 多副本检索优化 10
1.3 研究目标与内容 11
1.4 本文结构组织 11
1.5 本章小结 12
第 2 章 相关技术 13
2.1 Impala 实时查询系统 13
2.1.1 Impala 架构 13
2.1.2 Impala 调度模块 15
2.1.3 Impala 统计信息及选择度计算 17
2.2 本章小结 17
第 3 章 基于副本选择的大数据实时查询处理并行调度方法 19
3.1 概述 19
3.2 基本定义 19
3.3 调度方法 21
3.3.1 适用于单表查询的调度方法 23
3.3.2 适用于多表查询的调度方法 29
3.4 调度方法对副本一致性的影响 32
3.5 本章小结 32
第 4 章 Impala 查询处理代价模型 33

4.1 概述 33
4.2 操作模型 33
4.3 代价模型 34
4.4 中间结果估计 38
4.5 本章小结 39
第 5 章 系统实现 40
5.1 系统架构 40
5.2 模块实现 40
5.2.1 并行调度方法模块 40
5.2.2 代价模型模块 41
5.2.3 负载信息模块 41
5.2.4 统计信息模块 42
5.3 本章小结 42
第 6 章 实验评估 43
6.1 实验环境 43
6.2 实验设置 43
6.3 实验数据 45
6.4 实验结果与分析 46
6.4.1 *maxTime* 参数调优 46
6.4.2 *initialLoad* 参数调优 46
6.4.3 无并发查询请求情况下的适用于单表查询的调度方法对比实验 48
6.4.4 无并发查询请求情况下的适用于多表查询的调度方法对比实验 50
6.4.5 无并发查询请求情况下的不同副本数的调度方法对比实验 51
6.4.6 有并发查询请求情况下的调度方法对比实验 52
第 7 章 总结与展望 54
7.1 总结 54
7.2 展望 55
参考文献 56
攻读硕士学位期间主要的研究成果 61
致谢 62

图目录

图 2.1 Impala2.0 体系架构 14
图 3.1 查询计划树 20
图 3.2 基于副本选择的大数据实时查询处理流程 21
图 3.3 流网络 25
图 3.4 流 28
图 5.1 集成后的 Impala 2.0 体系架构 40
图 6.1 *maxTime* 对查询响应时间的影响 46
图 6.2 *initialLoad* 对查询响应时间的影响 47
图 6.3 单表查询响应时间对比 48
图 6.4 单表查询的读磁盘操作执行时间对比 49
图 6.5 多表查询响应时间对比 50
图 6.6 不同副本数下查询响应时间对比 52
图 6.7 并发查询处理情况下查询响应时间对比 53

表目录

表 1.1 查询处理并行调度问题分类 4
表 5.1 直方图 42
表 6.1 集群环境 43
表 6.2 不同数据集规模下数据库表及其总行数 45
表 6.3 不同数据集规模下 store_sales 表的数据量及其 HDFS Block 的数目 45
表 6.4 单表查询调度方法所需时间对比（ms） 49
表 6.5 多表查询调度方法所需时间对比（ms） 51

第1章 绪论

1.1 研究背景

在分布式数据库系统中，并行查询优化可分为一阶段和两阶段两大类方法[1]。所谓两阶段[2]，是指第一个阶段产生查询计划树，第二个阶段完成该计划树的并行处理调度。一阶段方法将两个阶段整合在一起，其解空间非常大。事实上，求第一个阶段的最优解是 NP 问题[3]。求第二个阶段的最优解的问题经过简化以后也是 NP 问题[4]。一阶段方法的搜索空间要远大于两阶段方法。因此，一阶段方法只是在理论上可以找到全局最优解。

查询处理并行调度问题的研究已经有几十年的历史了，对于单查询任务而言，调度的目标就是最小化响应时间。对于多查询任务而言，调度的目标就是提升整个系统的吞吐量[5]。单查询任务的并行调度问题的核心是并行是有代价的。并行度越高通信代价就越大。单个查询任务的并行调度问题已经是一个难解问题，如果不考虑集群中各个节点之间的通信代价，那么单个查询任务的并行调度问题就简化为多处理节点调度问题[6]。而多处理节点调度问题是 NP 问题。多查询任务的并行调度问题在单查询任务并行调度问题的基础上，还需要考虑各个处理节点已有的查询任务，因而更加复杂。

查询处理并行调度包含任务模型、代价模型和调度方法三个部分。

任务模型包含操作之间的并行性和操作内部的并行性。操作树的一个结点代表一个操作。操作之间的并行性有三种：流水线并行、阻塞并行和独立并行。独立并行指操作之间没有输入输出关系。当操作之间有输入输出关系时，有阻塞和流水线并行两种。每一个操作都有 0 个或多个输入集合，有一个输出集合。操作的输入输出均有阻塞和流水线两类。阻塞输入是指操作在获得整个输入集合之后才会开始执行。流水线输入是指一旦操作获得输入元组就直接开始执行。阻塞输出是指操作结束时产生整个输出集合。流水线输出是指操作在执行过程中不断有元组产生。只有流水线输入操作和流水线输出操作之间才是流水线并行关系，其

他操作之间均为阻塞关系。操作树的边表示操作之间的数据流，因而，操作树的边分为流水线边和阻塞边两类。如果一个操作被划分为多个子操作，那么多个子操作之间的并行性就是操作内部的并行性。

代价模型定义并计算查询处理的代价。代价越小表示调度越优。分布式查询处理的代价主要包含两个部分：本地的处理代价和通信代价。分布式查询处理的代价受底层架构影响。底层架构主要有共享内存、共享磁盘和无共享三类[7]。若采用共享内存的架构，则处理器之间的通信代价远小于本地的处理代价。因此，该架构的代价只包含本地的处理代价。若采用共享磁盘的架构，则将数据库表所需数据从磁盘读入内存的操作不存在并行调度问题。无共享架构又叫做 MPP（Massively Parallel Processing）架构，其具有廉价、高扩展、高可用的优点，其上的代价需要考虑本地的处理代价和通信代价两个部分。本课题只研究该架构上的代价模型。

调度方法按照是否在运行期动态调整，分为自适应的调度方法和非自适应的调度方法两类。自适应的调度方法会在操作执行的过程中重新优化调度策略。非自适应的调度方法在操作执行之前确定调度策略，在操作执行的过程中不再改变调度策略。调度方法需要考虑操作的并行度，以及为操作选择处理节点。在为操作选择处理节点的过程中需要考虑负载均衡。负载均衡包括操作之间的负载均衡和操作内部的负载均衡。操作之间的负载均衡实现比较困难[8]。主要原因有：第一，并行度和处理节点的分配都是在并行处理调度阶段完成的，其所基于的代价模型很可能是不准确的；第二，因为处理节点和操作都是离散的实体，所以在将处理节点分配给操作时存在离散误差；第三，存在流水线延迟问题，即查询树顶端的操作，也就是最后一个执行的操作，必须等其他操作完成，才能开始执行。操作内部的负载均衡需要考虑数据倾斜的问题。

随着数据的爆炸式增长，大数据时代已经来临。2010 年，Google 公司在一篇论文中介绍了可扩展、交互式的实时查询系统 Dremel[9]。Dremel 能够在秒级时间内分析 PB 级别的数据。除此之外，现有的大数据实时查询系统包括 Cloudera Impala[10]、Apache Spark[11]、Apache Drill[12]等。其中，Cloudera Impala 使用 MPP

计算框架，比使用 MapReduce[13]计算框架的数据仓库工具 Apache Hive[14]要快 3~90 倍。因其性能出众，Cloudera Impala 引起了业界的高度重视。其在 HDFS[15]上提供快速、交互式的 SQL 查询。HDFS 将文件分成多个数据块，并为每个数据块创建多副本。Impala HDFS 中的每个 DataNode 上均可启动 Impala 服务进程，用户可连接到任一 Impala 服务进程，提交查询请求并获得查询结果。在查询处理的过程中，该查询服务进程首先解析查询语句，然后生成查询计划树，之后制定调度方案，最后返回查询结果。Impala 目前的并行调度策略为选择已分配任务最少的节点。这种方法虽然可以保证各个节点上的任务量不会严重倾斜，但任务越均衡，查询的响应时间不一定越小。

例如，集群$\{n_1, n_2, n_3, n_4, n_5\}$，每个节点上有 1 个磁盘，即共有 5 个磁盘$\{d_1, d_2, d_3, d_4, d_5\}$。节点 n_5 是 NameNode，其余节点均为 DataNode。每个 DataNode 上运行 1 个 Impala 服务进程。用户连接节点 n_1 上的 Impala 服务进程，提交查询请求“SELECT * FROM A, B where A.t = B.t”。数据表 A 由数据块 b_1 构成，数据表 B 由数据块 b_2 构成，数据块大小均为 128MB。b_1 在磁盘$\{d_1, d_2, d_3\}$上有副本。b_2 在磁盘$\{d_2, d_3, d_4\}$上有副本。所有磁盘的读取速率为 128MB/s，网络传输速率为 64MB/s。以任务量均衡或负载均衡为优化目标的调度策略是相同的，即 n_1 执行将 b_1 从磁盘读入内存的任务，n_2 执行将 b_2 从磁盘读入内存的任务，若采用该调度策略，则查询处理 1s 之后，b_1 和 b_2 被读入内存。3s 之后，b_2 被发送到 n_1，开始执行连接操作。若为数据块 b_1 和 b_2 均选择磁盘 d_1 上的副本，则查询执行 2s 之后，b_1 和 b_2 均被读入内存，开始执行连接操作。因此，为数据块选择副本的过程中，若以任务量均衡或负载均衡作为优化目标，不能有效的提升大数据实时查询系统的性能。

和 Impala 类似，Pivotal 公司的 HAWQ[16]也采用 MPP 计算框架并在 HDFS 上提供快速、交互式的 SQL 查询。此外，Pivotal 公司的 Greenplum Database[17]是 MPP 架构的关系型数据库。Solimen 等人[18]设计 Orca 查询优化器，其适用于 Privotal 公司的所有数据管理产品，但是，Orca 并未涉及副本选择的并行查询调度问题。

因此，我们将基于已有的工作，从任务模型、代价模型和调度方法三个方面对查询处理并行调度进行研究，并在大数据实时查询平台 Impala 上进行实现，以提高 Impala 查询处理能力。

1.2 课题研究国内外现状

查询处理并行调度主要由任务模型、代价模型和调度方法三部分组成。因为多副本检索优化问题和基于副本选择的大数据实时查询处理并行调度问题有相似之处，所以下面将分别介绍这几方面相关的研究工作。

1.2.1 任务模型

Chekuri 等人[4]按照任务模型对查询处理并行调度问题进行了分类，如表 1.1 所示：

表 1.1 查询处理并行调度问题分类

	操作内部无并行	操作内部有并行
操作树中仅有流水边	流水线型操作树调度（POT）	操作内部有并行的流水线操作树调度（POTP）
操作树中仅有阻塞边	阻塞操作树调度（BOT）	操作内部有并行的阻塞操作树调度（BOTP）
操作树中既有流水边又有阻塞边	操作树调度（OT）	操作内部有并行的操作树调度（OTP）

Hasan 等人[2]对星形和线性结构的 POT 问题构建了并行模型。Chekuri 等人[4]在的基础上对任意结构的 POT 问题构建了并行模型。但是，这两类并行模型都没有考虑操作之间的独立并行性和操作内部的并行性。Garofalakis 等人[5]的并行模型考虑到了操作之间的并行和操作内部的并行。其中，操作之间的并行考虑到了流水线型并行和独立并行。

1.2.2 代价模型

商业数据库中的优化器所使用的代价模型往往并不考虑并行处理和传输的

代价，例如 Sophie 等人[19]主要以连接查询产生的中间结果作为代价模型。查询处理的代价可以用总时间或者响应时间来表示。总时间就是各个部分时间的总和。响应时间即从开始查询处理到完成查询的时间间隔。若以总时间表示查询处理的代价，则优化目标为系统的吞吐量最大。响应时间最小并不等价于总时间最小[20]。现有的代价模型分为两类：基于代价函数的代价模型和采用机器学习的代价模型。基于代价函数的代价模型用代价函数表示查询处理的代价。基于机器学习的代价模型将数据库系统当作黑盒，通过查询语句构成的测试集学习出代价模型。

1.2.2.1 基于代价函数的代价模型

基于代价函数的代价模型用代价函数表示查询处理的代价。代价函数既可以计算总时间，也可以计算响应时间。总时间的计算见公式（1.1）[6]：

$$T_{total} = T_{CPU} \times \#insts + T_{I/O} \times \#I/Os + T_{MSG} \times \#msgs + T_{TR} \times \#bytes \quad \text{公式（1.1）}$$

其中，T_{CPU} 表示执行一个 CPU 指令所需时间，$T_{I/O}$ 表示一次磁盘 I/O 所需时间，T_{MSG} 表示生成和接受一个消息所需时间，T_{TR} 表示在节点之间转移 1B 数据量所需时间。*#insts* 表示处理该查询所需指令的数目，*#I/Os* 表示处理该查询所需磁盘 I/O 数目，*#msgs* 表示处理该查询所需消息数目，#bytes 表示处理该查询所需转移数据的总字节量。因为获得该查询所需指令的数目、磁盘 I/O 的数目、消息数目和转移数据的总字节量的代价较大，所以公式（1.1）不适用于实际代价模型。实际的代价模型只是选择对总时间影响较大的部分作为代价。例如，因为 Hive 查询处理的中间结果会写入 HDFS 中，即存在大量的写入和读取的操作，所以磁盘 I/O 所需时间对总时间影响较大。因此，Tomasz、Okcan、Herodotou 和 Anja 等人[21][22][23][24]均以 I/O 操作次数作为查询处理的代价。该类代价模的不足是只适用于 I/O 开销占总时间比重较大的情况，例如 MapReduce 计算框架。事实上，MPP 框架下的中间结果的写入和读取操作的次数明显少于 MapReduce 计算框架，I/O 开销占总时间的比重也会下降。

响应时间的计算见公式（1.2）[25]：

$$T_{response} = T_{CPU} \times seq\_\#insts + T_{I/O} \times seq\_\#I/Os + T_{MSG} \times seq\_\#msgs + T_{TR} \times seq\_\#bytes$$

公式（1.2）

其中，T_{CPU}、$T_{I/O}$、T_{MSG}和 T_{TR}的含义和公式（1.1）中的含义相同。*seq_#insts* 表示必须顺序执行的指令数，*seq_#I/Os* 表示必须顺序执行的磁盘 I/O 数，*seq_#msgs* 表示必须顺序处理的消息数，*seq_#bytes* 表示必须顺序转移的数据的字节量。因为获得 *seq_#insts*、*seq_#I/Os*、*seq_#msgs* 和 *seq_#bytes* 的代价较大，所以公式（1.2）不适用于实际代价模型。

Ganguly 等人[26]定义估计函数 G 和 H。在 90%~95%的查询情况下，该代价函数的误差为 20%~60%。其假设所有查询操作在任何处理节点上的顺序处理时间是已知的。然而，该时间需要估计。Shankar 等人[27]定义数据移动操作的代价估计函数。其假设在该查询处理的过程中，系统不会处理其他查询。因为大数据实时查询系统支持并发的查询处理，所以该假设不成立。Sampanio 等人[28]为对象数据库的读磁盘、选择和连接操作定义了代价计算函数。其不足之处在于只适用于对象数据库。

为了提高基于代价函数的代价模型的准确性，需要估计查询处理的中间结果的大小，包括抽样方法[29][30][31]、参数化方法[32][33]、直方图方法[34][35]。因为抽样方法通常需要通过抽样获得大量的样本以保证代价估计的准确性，所以基于抽样方法的代价估计开销会较大。参数化方法的优点是当假设的数学分布与真实数据分布比较接近时，其具有较高的估计准确性。其不足之处在于当假设的数学分布和真实数据分布存在较大偏差时，其具有较低的估计准确性。与抽样方法和参数化方法相比，直方图方法的优点在于运行时开销小，无需假设数学分布，通过耗用少量的空间便可达到较高的估算准确性。Poosala 等人[36]通过对之前的直方图（包括 Trivial 直方图、等宽直方图、等高直方图、V-Optimal(*F*, *F*)直方图和 V-Optimal-End-Biased(*F*, *F*)直方图）分类研究，提出组合方式直方图（前几种直方图的组合与改进）。其实验表明 Maxdiff(V, A)直方图和 V-Optimal(V, A)直方图的错误率最低为 0.77。Poosala 等人推荐 Maxdiff(V, A)直方图，认为其构造时间复杂度和准确度均接近最优的直方图。

1.2.2.2 基于机器学习的代价模型

基于机器学习的代价模型将数据库系统当作黑盒，通过查询语句构成的测试集学习出代价模型。

Ganapathi 等人[37]首先抽取查询请求的特征，主要是操作的数目和表的基数，然后用核典型相关分析（Kernel Canonical Correlation Analysis）方法建立查询请求的特征空间和性能空间之间的映射。性能空间包括查询响应时间等。因为其基于 k 近邻方法，所以其预估的处理时间不会超过训练集中最大的处理时间。因此，若某个查询请求的处理时间远大于训练集中的查询请求的处理时间，则该代价模型无法保证预测的准确性。

Akdere 等人[38]使用了和 Ganapathi 等人[37]相似的方法。不同点是使用支持向量机（Support Vector Machine）替换核典型相关分析，并且应用于操作级别。但是 Akdere 等人自己的实验表明，工作量的变化对该方法影响较大。

Duggan 等人[39]考虑到查询并发处理对单个查询请求的响应时间的影响并采用多元线性回归模型（Linear Multivariate Regression Model）。但是，其假设所有的查询请求都是已知的。实际中很难满足这一点。

基于机器学习的代价模型主要存在 3 个不足：第一，代价模型的准确性依赖于由大量的查询语句构成的测试集。对于商业数据库而言，获得由大量的，真实用户提交的查询语句构成的测试集并不是难事。但是对于实验室而言，获得好的测试集比较困难。第二，因为将数据库当作黑盒，所以学习得到的代价模型不具有解释性。

1.2.3 调度方法

调度方法按照是否在运行期动态调整，分为自适应的调度方法和非自适应的调度方法两类。自适应的调度方法是指在查询处理的过程中动态调整调度策略。非自适应的调度方法是指在查询处理前确定调度策略，在查询处理的过程中，调度策略不变。

1.2.3.1 非自适应的调度方法

1) 操作树划分

由 1.2.1 节知，按任务模型划分，POT 问题是查询处理并行调度问题的一种，针对该类问题的算法主要有三步，第一步把操作树转化为单调树；第二步将单调树划分为多个分片；第三步将分片分配给处理节点。LPT 算法是适用于独立并行调度的简单启发式算法，Hasan 等人[2]指出由于没有考虑通信代价，LPT 算法对于 POT 问题并不是很有效。Hasan 等人通过贪心策略去掉一些无意义的调度，形成修改后的 LPT 算法。Hasan 等人还提出了寻找最优连接调度的 BalancedCuts 算法，并将 BlancedCuts 算法和 LPT 结合形成了 Hybrid 算法。Temehchi 等人[40]研究了 Hybrid 算法在 2 处理节点情况下的调度响应时间的最大值。Odubasteanu 等人[41]对 BalancedCuts 算法进行优化，提出了 OptimalBalancedCuts 算法，并将 OptimalBalancedCuts 算法和 LPT 结合形成了 OptimalHybrid 算法。Checuri 等人[4]提出了局部最优的 LocalCuts 算法，以及针对 LocalCuts 出现通信过高的调度，提出了 BoundedCuts 算法。Temehchi 等人研究了 LocalCuts 和 BoundedCuts 算法在多处理节点情况下的调度响应时间的最大值。Odubasteanu 等人对 LocalCuts 和 BoundedCuts 算法的参数进行了调优。Hsiao 等人[42]提出了基于同步执行时间的算法，该算法只适用于在并行集群中调度连接操作，无法扩展到整棵操作树的调度。操作树划分的方法的不足之处在于其只适用于 POT 问题，并不适用于其他任务模型。例如，Impala 的查询处理并行调度问题的任务模型为 BOTP。

2) 粗粒度并行

Ganguly 等人[43]在假设通信代价小于处理代价的 f（0<f<1）倍的基础上，提出了粗粒度并行问题，进而提出适用于该问题的调度方法。该算法将所有操作分为两种，即可移动操作（如连接操作）和不可移动操作（如读磁盘操作）。考虑到了操作之间有流水线型并行和独立并行。Garofalakis 等人[5]将粗粒度并行问题转化为集装箱问题，进而采用表调度方法。粗粒度并行忽略网络传输的代价。然而，在分布式环境下，网络传输的代价不能忽略。

3) 启发式

Rahm 等人[44]提出的简单启发式调度方法主要包括 MIN-IO、MIN-IO-SUOPT

和 OPT-IO-CPU。其均尽量避免临时文件 I/O。因为其考虑到系统当前状态，所以可以实现动态负载均衡。但是负载越均衡，查询的响应时间不一定越短。1.1 节的例子说明了这一点。Nam 等人[45]提出的调度方法综合考虑了 Cache 命中率和服务节点的负载状态。该调度方法假设单个节点可以处理查询请求，即未考虑多个节点协作完成查询请求的情况。例如，在 Impala2.0 系统中，大部分情况下由多个节点协作完成一次查询处理。Kolalei 等人[46]提出使用蚁群算法解决并行查询优化的调度问题。该方法的主要缺点是算法的时间复杂度较高。

1.2.3.2 自适应的调度方法

自适应的调度方法会根据查询处理过程中查询任务的中间结果，以及系统当前的状态调整调度策略，避免因各个处理节点负载不平衡而对并行处理产生影响。

Hong 等人[47]将任务分为“IO-bound”和“CPU-bound”，通过在处理的过程中自适应地调节任务的并行度以保证系统工作在 IO-CPU 平衡点。该方法已集成到系统 XPRS[48]中。

Kabra 等人[49]在查询处理过程中收集统计信息，如果统计信息和编译期的估计误差较大，就重新优化查询中还未处理的部分。因为重新优化查询中还未处理的部分比较困难，所以先将已完成的部分转化为临时表，之后生成基于临时表的查询语句。该查询语句等效于原查询语句，并被重新处理。该方法已集成到系统 Paradise[50]中。

Rahm 等人[51]主要考虑操作之间的负载均衡，其将并行度选择和处理节点选择分开考虑，提出了 DJP+ ALUP 算法，其中 DJP 时间复杂度为 O(1)。其在处理过程中根据 CPU 利用率动态调整并行度。因为周期性获得 CPU 利用率存在阶段性负载信息过时问题，所以，ALUP 算法人工更新 CPU 利用率。但在实际中，一个任务对 CPU 利用率的影响很难量化。而且其也没有从理论上对动态调整的公式进行分析。

Mehta 等人[52]提出 Rate Match 算法，主要考虑操作之间的负载均衡。该算法

通过操作之间的处理速度相匹配，避免出现发送元组的操作阻塞和分配给接收元组的操作的处理节点空闲的情况。该算法的不足之处在于只适用于操作之间为流水线并行关系。

因为不同种类的数据倾斜会对并行连接操作产生不同的影响[53]。DeWiit 等人提出使用多种连接算法[54]，不同的算法适用于不同程度的数据倾斜。然后在运行期间由调度器决定哪一个算法是最好的。其主要考虑操作内部的负载均衡。

Shah 等人[55]提出了一种称为 Flux 的调节任务分配的操作。监控每个节点在运行期的执行速度和空闲时间，并相应的移动任务。由于任务移动之前可能已经处理了一部分，移动任务也需要移动任务的状态，移动代价很大。而且该方法只适用于流式处理系统。

因为副本选择需在查询处理之前确定，所以自适应的调度方法并不适用于多副本条件下使用 MPP 计算框架的查询处理并行调度问题。

1.2.4 多副本检索优化

所谓多副本检索优化问题是指若一个文件分成多个块，每个块在多个磁盘上有副本，则如何选择副本可以最小化检索所需时间。Chen 等人[56]提出将多副本检索优化问题转化成最大流问题，并采用 Ford-Fulkerson 算法解决该问题。其主要考虑单节点上的同构磁盘阵列。Tosun 等人[57]将多副本检索优化问题进行扩展，考虑到双节点上的同构磁盘阵列并且针对扩展后的问题提出了一种黑盒的最大流算法。Altiparmak 等人[58]对多副本检索优化问题进一步泛化。其考虑多个节点的磁盘阵列异构和每个磁盘的初始负载。对比了 Ford-Fulkerson 方法和压入重标记算法。实验表明压入重标记算法的算法复杂度要低于 Ford-Fulkerson 方法。之后，其提出 PushRelabelBinary 算法。在最坏情况下，该算法的时间复杂度为 $O(\log|Q|*|Q|^3)$，其中$|Q|$表示检索的 bucket 的数目。

多副本检索优化问题和大数据实时查询处理并行调度问题既有相同点，又有不同点。相同点是文件分块，且每个块均有多个副本，目标也均为响应时间最小。不同点是多副本检索优化问题并不包含连接操作，但是，在大数据实时查询处理

并行调度问题中，含有连接操作的查询请求在所有查询请求中所占比重较高。

1.3 研究目标与内容

本课题“基于副本选择的大数据实时查询处理并行调度研究与实现”的主要研究对象是大数据实时查询处理并行调度，根据 1.2 节中对调度方法的研究可以看出，现有的调度方法未考虑数据文件分块及每个块均有若干个副本的情况。为此，本人选择“基于副本选择的大数据实时查询处理并行调度研究与实现”作为毕业论文的课题，其目的是以此课题为切入点，通过阅读相关的会议论文、期刊论文和书籍等资料进行大数据实时查询处理和分布式数据库查询处理并行调度理论知识的学习，从而对大数据实时查询处理并行调度问题进行初步探索并将本课题提出的并行调度方法集成到大数据实时查询系统 Impala2.0。

基于以上研究目的，本文提出的研究目标如下：

1) 建立多副本条件下使用 MPP 计算框架的查询处理并行调度问题模型；
2) 提出基于副本选择的大数据实时查询处理并行调度方法；
3) 在大数据实时查询系统 Impala2.0 上实现该调度方法；
4) 设计实验对比大数据实时查询系统 Impala2.0 改进前后的性能。

针对上述研究目标，本文首先研究了并行调度任务模型、代价模型和并行调度方法三方面的相关工作，同时深入分析了大数据实时查询系统 Impala2.0 的体系架构，分析了大数据实时查询系统 Impala2.0 的并行调度模块和大数据实时查询系统 Impala2.0 的统计信息及选择度计算模块，然后提出基于副本选择的大数据实时查询处理并行调度方法，接着构建 Impala2.0 查询处理代价模型并将调度方法集成到 Impala2.0，最后，对提出的基于副本选择的大数据实时查询处理并行调度方法的进行实验验证，并对方法的有效性和复杂度进行分析。

1.4 本文结构组织

本文共分为 7 章。第 1 章为绪论，主要介绍了分布式数据库系统查询处理并行调度的研究背景、大数据实时查询系统的发展背景和现有大数据实时查询系统的不足，综合介绍了国内外关于查询处理并行调度在任务模型、代价模型、并行

调度方法三个方面和与多副本检索优化相关的研究工作，分析了现有方法的优点和不足之处，最后介绍了本课题的研究目标和内容及本文的组织结构。

第 2 章为相关技术，首先介绍了大数据实时查询系统 Impala2.0 的体系架构，然后介绍了大数据实时查询系统 Impala2.0 的调度模块，最后介绍了大数据实时查询 Impala2.0 的统计信息及选择度计算模块。

第 3 章为基于副本选择的大数据实时查询处理并行调度方法，是本文的主要内容。首先概要地介绍基于副本选择的大数据实时查询处理并行调度方法，然后定义相关的基本概念，最后详细阐述了基于副本选择的大数据实时查询处理并行调度方法。

第 4 章为 Impala 查询处理代价模型，也是本文的主要内容。本章首先为 Impala 查询处理过程中的常见操作构建模型，然后详细介绍了本课题提出的 Impala 查询处理代价模型，最后简要介绍了本课题如何进行中间结果估计以提高 Impala 查询处理代价模型的准确度。

第 5 章为系统实现，本章首先介绍集成了基于副本选择的大数据实时查询处理并行调度方法的 Impala2.0 系统总体框架，然后分别介绍了并行调度方法模块、代价模型模块、负载信息模块、统计信息模块的具体实现。

第 6 章为实验评估，本章首先介绍实验环境，然后介绍实验设置，接着介绍实验数据，最后对实验结果进行了详细的分析说明。

第 7 章为总结和展望，总结本课题的研究工作，同时针对本课题工作的不足，对后续的工作进行展望。

1.5 本章小结

本章首先介绍了本文的研究背景及意义，对现有的分布式数据库系统中的并行调度和大数据实时查询系统进行了简要的描述；其次综合介绍了国内外关于查询处理并行调度在任务模型、代价模型、并行调度方法三个方面和与多副本检索优化相关的研究工作；然后提出了本课题的研究内容及目标；最后介绍了本文的组织结构。

第2章 相关技术

2.1 Impala 实时查询系统

由 Cloudera 公司开发的大数据实时查询系统 Impala 对存储在 HDFS 和 HBase 上的数据进行快速的、交互式的 SQL 查询。本课题主要研究 Impala2.0。Impala 使用和 Apache Hive 相同的元数据、SQL 语法、ODBC 驱动器以及用户界面。Impala 的优点主要包括：

1） 使用科学家和数据分析师熟悉的 SQL 接口；

2） 能够交互式的查询存储在 Apache Hadoop 中的大数据；

3） 分布式和并行化的查询处理能充分利用集群资源；

4） 无需拷贝或导入导出就能够在不同的组件间共享数据文件。

2.1.1 Impala 架构

Impala2.0 是分布式的，MPP(Massively Parallel Processing)结构的数据库系统。其体系架构如图 2.1 所示。其中，黄色矩形框表示的是 Impala2.0 系统的重要组件。蓝色矩形框表示的是和 Impala2.0 系统交互的其他组件。Impala2.0 系统的核心为查询处理层。其接收查询应用层发出的查询请求，解析查询请求，完成具体的执行操作，并将查询结果返回给查询应用层。在查询处理过程中，元数据服务层为查询处理层提供元数据信息。Impala2.0 系统的服务端主要由以下三个部分组成：

1） Impala 服务进程

Impala 服务进程可以从 HDFS 中读取数据，也可以将数据写入 HDFS 中。用户可通过 Impala-shell 命令、Hue、JDBC 或者 ODBC 向 Impala 服务进程提交查询请求。Impala 服务进程包括查询计划器、查询协调器和查询执行器，是 Impala2.0 的核心部分。查询计划器对用户提交的查询语句进行解析并生成查询计划。查询计划器首先生成查询计划树，查询计划树由 Plan Node 构成。常见的 Plan Node 包括 Scan Node（读磁盘操作）、Join Node（连接操作）、Exchange Node（数据转

移操作）和 Aggregation Node（聚集操作）构成，然后将查询计划树分为若干个 PlanFragment，一个 PlanFragment 是查询计划树的子树且其叶子节点必定是 Scan Node。查询协调器根据查询计划以及集群状况分配查询任务，接收查询执行器返回的中间结果，形成最终查询结果并返回给用户。查询协调器包括调度模块，其根据查询计划以及集群状况制定调度策略。查询执行器负责具体的读磁盘操作（将数据从磁盘读入内存的操作）、选择操作、连接操作、投影操作和聚集操作。Impala 服务进程会通过 StateStore 服务进程了解集群中的其他 Impala 服务进程的健康状况，通过 Catalog 服务进程了解数据库的元数据信息。

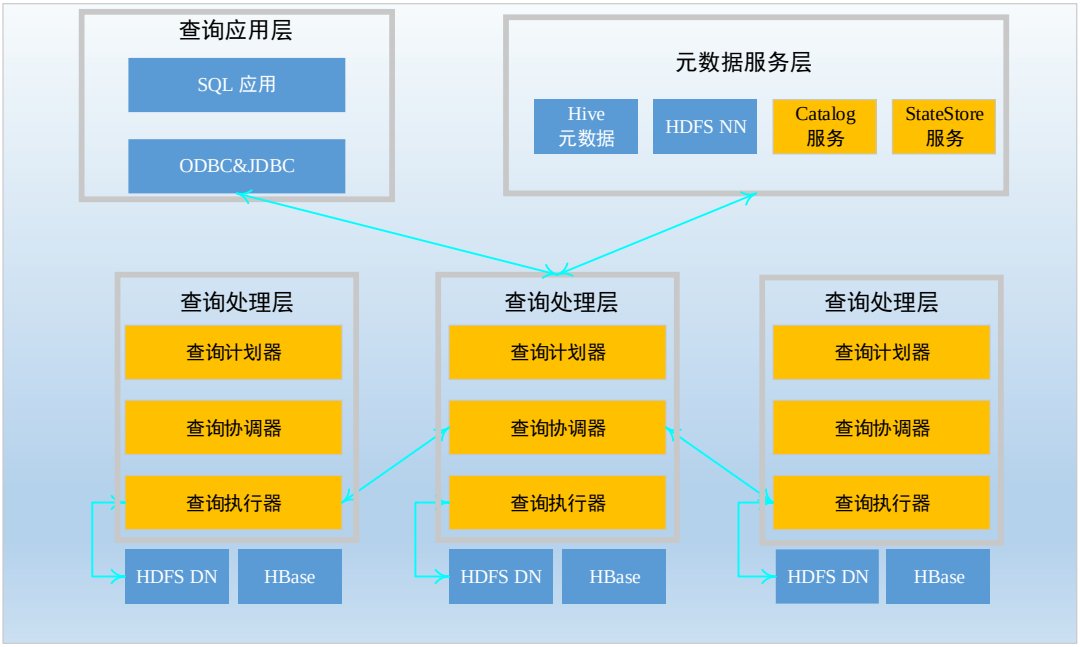

图 2.1 Impala2.0 体系架构

2） StateStore 服务进程

StateStore 服务进程监控所有的 Impala 服务进程的健康状况。当某个 Impala 服务进程由于硬件错误、网络错误、软件问题或其他原因无法工作时，StateStore 服务进程会将该消息通知其他 Impala 服务进程，以避免将查询处理任务分配给该 Impala 服务进程。若未运行 StateStore 服务进程或者 StateStore 服务进程不可访问，Impala 服务进程仍然可以运行并且 Impala 服务进程之间仍然可以相互通信。但是，整个系统的鲁棒性会下降。其原因是若某个 Impala 服务进程出了问题，其他 Impala

服务进程并不知道，会将查询任务分配给他，结果通信时才发现该 Impala 服务进程无法工作，整个查询的执行会失败。不过，当 StateStore 服务进程再次运行时，其会和所有 Impala 服务进程重新通信，以监控整个集群的健康状况。

3） Catalog 服务进程

Catalog 服务进程管理数据库的元数据信息。因为 Impala 服务进程会缓存元数据信息，所以当某个 Impala 服务进程处理完 DDL（Data Definition Language）或 DML（Data Manipulation Language）语句时，其他 Impala 服务进程缓存的元数据信息已经过时。若无 Catalog 服务进程，则用户必须手动更新元数据信息。若由于 CREATE DATABASE, DROP DATABASE, CREATE TABLE, ALTER TABLE, 或 DROP TABLE 语句造成元数据信息过时，则需用户输入 INVALIDATE METADATA 语句更新元数据信息。若由于 INSERT 语句造成元数据信息过时，则需用户输入 REFRESH 语句更新元数据信息。若启动 Catalog 服务进程，则 Impala 服务进程可通过 Catalog 服务进程获得最新的元数据信息。不过，对于 Hive 完成的 DDL 和 DML 操作，仍需用户输入 INVALIDATE METADATA 或 REFRESH 语句更新元数据信息。

2.1.2 Impala 调度模块

Impala2.0 提供纯虚类 Scheduler 以规范调度模块的接口。Impala2.0 提供了一个派生类 SimpleScheduler 完成具体的调度。其根据查询计划和本次查询涉及的数据表文件的分布情况制定调度策略。一个 Scan Node 表示对一张数据库表的读磁盘操作，一张数据库表以文件的形式存储在 HDFS 中。因为 HDFS 将一个文件分为多个 HDFS Block，所以每个 Scan Node 对应若干个 Scan Range，一个 Scan Range 对应一个 HDFS Block。因为 HDFS 为每个 HDFS Block 创建多个副本，所以一个 Scan Range 对应多个 Location，每个 Location 对应副本所在的节点以及磁盘。

SimpleScheduler 制定调度策略的过程包括为每个 Scan Range 选择执行节点和为每个 PlanFragment 选择执行节点两步。若不考虑缓存，则为每个 Scan Range 选择执行节点的流程如下所示：

1） 判断是否还有数据表未选择副本，若是，转步骤 2），若否，则结束；

2） 获得该数据表的所有 Scan Range；

3） 判断是否还有 Scan Range 未选择副本，若是，转步骤 4），若否，转步骤 1）；

4） 为该 Scan Range 选择副本，转 3）。

a) 获得该 Scan Range 的所有副本所在节点列表；

b) 若节点上没有 Impala 服务进程，则其初始已分配字节量为 2^{63}-1B；否则，其初始已分配字节量为 0B；

c) 若没有该节点已分配字节量的记录，则其已分配字节量为初始已分配字节量；

d) 选择已分配字节量最小的节点；

e) 若选择的副本所在节点没有 Impala 服务进程，则需为该副本选择其他节点上的 Impala 服务进程远程读取数据，Impala 会轮询所有有 Impala 服务进程的节点；

f) 更新该节点已分配字节量。

因为相比于本地读取，远程读取的代价更高，所以需避免远程读取。通过初始化没有 Impala 服务进程的节点的已分配字节量为 2^{63}-1B，有 Impala 服务进程的节点的已分配字节量为 0B，SimpleScheduler 就可实现在副本选择时优先考虑有 Impala 服务进程的节点的目的。

因为 PlanFragment 的叶子节点是 Scan Node，所以 PlanFragment 的执行节点和 Scan Node 的执行节点相同。Scan Node 的执行节点由其包含的 Scan Range 的执行节点决定。

SimpleScheduler 采用了贪心策略进行调度，即每次选择已分配字节量最小的节点。其实质是以任务量均衡为目标，但任务量越均衡，查询响应时间不一定越短，1.1 节的例子说明了这一点。

2.1.3 Impala 统计信息及选择度计算

虽然 Impala2.0 在其调度模块中并未使用代价模型，但其为了优化查询计划树，仍然收集统计信息。表级别的统计信息包括表的大小和表的行数。列级别的统计信息包括相异值的个数、空值的个数、该列所占字节量的平均值和最大值。

所谓选择度，是指满足条件的行在所要查询的对象集合中所占的比例，其中对象集合可以是表、视图或者中间结果集合，一般用 selectivity 表示。因为 Impala2.0 的统计信息有限，所以其无法准确计算选择度，具体如下：相等的选择度可由相异值的倒数得到。不等、小于、小于等于、大于等于和大于等二元谓词的选择度均为 0.1。若用 selectivity(p_1)表示谓词 p_1 的选择度，用 selectivity(p_2)表示谓词 p_2 的选择度，则谓词 p_1 和谓词 p_2 相与的选择度，即 selectivity(p_1 and p_2)可由公式（2.1）计算：

$$\text{selectivity}(p_1 \text{ and } p_2) = \text{selectivity}(p_1) \times \text{selectivity}(p_2) \quad \text{公式（2.1）}$$

谓词 p_1 和谓词 p_2 相或的选择度，即 selectivity(p_1 or p_2)可由公式（2.2）计算：

$$\text{selectivity}(p_1 \text{ or } p_2) = \text{selectivity}(p_1) + \text{selectivity}(p_2) \quad \text{公式（2.2）}$$

不满足谓词 p_1 的选择度，即 selectivity(not p_1)可由公式（2.3）计算：

$$\text{selectivity}(\text{not } p_1) = 1 - \text{selectivity}(p_1) \quad \text{公式（2.3）}$$

属性 a 介于 n_1 和 n_2 之间的选择度，即 selectivity(a between n_1 and n_2)可由公式（2.4）计算：

$$\text{selectivity}(a \text{ between } n_1 \text{ and } n_2) = \text{selectivity}(a >= n_1) \times \text{selectivity}(a <= n_2)$$

公式（2.4）

属性 a 不介于 n_1 和 n_2 之间的选择度，即 selectivity(a not between n_1 and n_2)可由公式（2.5）计算：

$$\text{selectivity}(a \text{ not between } n_1 \text{ and } n_2) = \text{selectivity}(a < n_1) + \text{selectivity}(a > n_2))$$

公式（2.5）

2.2 本章小结

本章首先介绍了 Impala 实时查询系统（Impala2.0）的体系架构，然后介绍了

其三个主要组成部分：Impala 服务进程、StateStore 服务进程和 Catalog 服务进程，接着介绍了 Impala 实时查询系统现有的调度模块，最后介绍了 Impala 实时查询系统现有的统计信息和选择度计算模块。

第3章 基于副本选择的大数据实时查询处理并行调度方法

3.1 概述

并行调度是并行查询优化的重要组成部分，并行调度方法一般由调度方法和代价模型两部分组成。本章将会对基于副本选择的大数据实时查询处理并行调度方法进行详细的阐述。基于副本选择的大数据实时查询处理并行调度方法将所有查询分为单表查询和多表查询两类。单表查询是指只涉及一张数据表的查询，而多表查询是指涉及多张数据表的查询。若是单表查询，则转化为多副本检索优化问题，应用最大流方法选择副本和执行节点；若是多表查询，则结合提出的 Impala 查询处理代价模型搜索近似最优调度。下一章会对 Impala 查询处理代价模型进行详细的阐述。

具体技术方案将在后续小节进行展开描述，在本章介绍具体方法之前，有必要首先介绍和基于副本选择的大数据实时查询处理并行调度方法相关的概念。

3.2 基本定义

定义 3.1 查询计划树。查询计划树 $T(V, E)$,，其中，V 是树中结点的集合，$E \subset V \times V$ 是树中边的集合。树中的结点代表操作，树中的边代表操作间的数据流。查询计划树的根结点用 $root(T)$表示，普通结点用 v 表示，包括 Scan Node、Join Node、Exchange Node 和 Aggregation Node 等。节点 v 的左孩子结点用 v_l 表示，右孩子结点用 v_r 表示。

例如，查询语句 select count(*) from a, b where a.t = b.t 所对应的查询计划树如图 3.1 所示。其中 Scan 结点代表将表的数据从磁盘读入内存的操作，Exchange 结点代表传输操作，Join 结点代表连接操作，Aggregation 结点代表聚集操作。

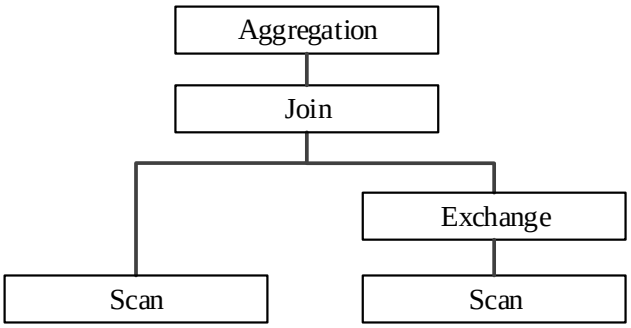

图 3.1 查询计划树

定义 3.2 查询处理并行调度。给定一棵查询计划树 $T(V, E)$，集群 $N=\{n_1, n_2, n_3, \cdots, n_m\}$，其中 m 为集群中节点数目，查询处理并行调度 $S=\{F_1, F_2, F_3, \cdots, F_m\}$，其中 F_m 是 V 的子集，表示需由集群中节点 n_m 处理的操作构成的集合，且 $F_1 \cup F_2 \cup F_3 \cup \cdots \cup F_m = V$。

定义 3.3 查询的响应时间。一个查询 Q 的响应时间 $t_{response}$ 是查询执行开始时刻 t_{start} 与得到查询结果时刻 t_{end} 的时间间隔，即

$$t_{response} = t_{end} - t_{start} \qquad \text{公式（3.1）}$$

定义 3.4 多副本条件下使用 MPP 计算框架的查询处理并行调度问题。

输入：查询计划树 $T(V, E)$，表的数据分布情况 D 和集群 $N=\{n_1, n_2, n_3, \cdots, n_m\}$。

输出：响应时间最小的调度，即使 $t_{response}$ 最小的查询处理并行调度 S。

多副本条件下的使用 MPP 计算框架的查询处理并行调度问题属于优化问题，优化目标为 $t_{response}$ 最小。

因为基于副本选择的大数据实时查询处理并行调度方法依赖于流网络，所以先定义流网络。

定义 3.5 流网络。流网络 $G=(V', E')$是一个有向图，其中，V'是流网络中顶点的集合，$E' \subset V' \times V'$是流网络中边的集合。其中任一条边$(u, w) \in E'$均有非负容量 $c(u, w) \geqslant 0$。若$(u, w) \notin E'$，则 $c(u, w) = 0$。流网络中存在两个特殊的顶点：源点 v_s 和汇点 v_e。

定义 3.6 流。流网络 G 中的流是一个实值函数 $f: V' \times V' \rightarrow R$. 其满足如

下三条性质：

容量限制：对于所有的顶点 $u, w \in V'$，要求 $0 \leqslant f(u, w) \leqslant c(u, w)$。

反对称性：对于所有的顶点 $u, w \in V'$，$f(u, w) = -f(w, u)$。

流量守恒：对于所有的顶点 $u \in V' - \{v_s, v_e\}$，有

$$\sum_{w \in V'} f(u, w) = 0 \qquad \text{公式（3.2）}$$

流的总量。流出源点的量表示流的总量，即，

$$|f| = \sum_{u \in V'} f(v_s, u) \qquad \text{公式（3.3）}$$

因为基于副本选择的大数据实时查询处理并行调度方法考虑到磁盘负载，所以先定义磁盘负载。

定义 3.7 磁盘负载。一个磁盘的负载 L 是从当前时刻 t_{now} 到其 I/O 任务完成时刻 t_{finish} 的时间间隔，即，

$$L = t_{finish} - t_{now} \qquad \text{公式（3.4）}$$

3.3 调度方法

基于副本选择的大数据实时查询处理流程如图 3.2 所示：

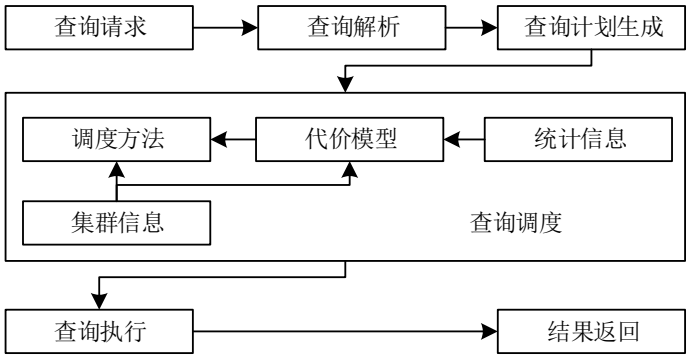

图 3.2 基于副本选择的大数据实时查询处理流程

首先解析用户提交的查询请求，然后生成由操作结点构成的查询计划树，接着应用本文提出的调度方法完成查询处理并行调度，最后执行并将结果返回。基于副本选择的大数据实时查询处理并行调度方法依赖于查询处理代价模型和集

群信息。查询处理代价模型依赖于统计信息和集群信息。第 4 章将详细介绍查询处理代价模型，集群信息包括磁盘读取速率，预计磁盘完成其上 I/O 任务的时刻以及集群网络传输速率。

基于副本选择的大数据实时查询处理并行调度方法的伪代码如算法 3.1 所示：

算法 3.1：ParallelSchedule

```
    // D 表示数据分布.
    // byte 表示一个 HDFS Block 的字节量
    // diskSpeed 表示所有磁盘的读取速率
    // netSpeed 表示网络传输速率
    输入：T, D, byte, diskSpeed, netSpeed
    输出：S
    // L 表示整个集群的负载信息
1   L ← GetLatestLoadInfo ();
2   if IsSingleTableQuery (T) then
3       G ← ConstructGraph (D);
        // R 表示副本选择的结果
4       R ← SRPushRelabelBinary (G, L, t, n, byte, diskSpeed, netSpeed)
        // S 表示调度
5       S ← ExecutionNodeSelection (T, R);
6       UpdateLoadInfo ();
7       return S;
8   end
9   else R ← OriginalSelectReplication (D);
10      S ← ExecutionNodeSelection (T, R);
        // minC 表示最小代价
11      minC ← CalculateCost (S)
        // Ri 表示关键操作的副本选择结果
        // I 表示执行关键操作的节点集合
12      Ri, I ← SelectReplicationImportant (D, L);
        // maxTime 表示搜索最优调度策略的次数
        // time 表示当前次数.
13      for time = 1 to maxTime do
            // L′ 表示整个集群的临时负载信息
14          L′ ← L;
            // Ru 表示非关键操作的副本选择结果.
15          Ru ← SelectReplicationUnimportant (D, I,
                                time, maxTime, L′ );
            //S'暂存临时调度策略
16          R ← Ru + Ri;
17          S' ← ExecutionNodeSelection (T, R);
            //C 暂存代价
18          C ← CalculateCost ( S′ );
19          if C < minC then
```

```
20              S ← S';
21              C ← minC;
22          end
23      end
24      UpdateLoadInfo ();
25      return S;
26  end
```

首先获得本次查询涉及的数据所在磁盘的负载信息 L，如行 1 所示。因为并行调度方法的执行时间是毫秒级的，所以本文假设在并行调度方法的执行过程中集群不会因其他任务而发生负载的变化。与多次获得负载信息的方法相比，本文的方法减少了网络传输的负载和方法的执行时间。因为不同负载的节点处理相同任务所需时间不同，所以与不考虑集群负载的调度方法相比，本文的方法将得到更优的调度策略，即使查询响应时间更低的调度策略。

然后将所有查询分为两大类：单表查询和多表查询。单表查询是指仅涉及一张数据表的查询。多表查询是指涉及多张数据表的查询。因为在单表查询的过程中，读磁盘操作所需时间占总查询时间的比重较大，由 Amdahl 定律[59]可知，减少该过程的时间可以有效减少响应时间，所以在单表查询的情况下，可以将多副本条件下的使用 MPP 计算框架的查询处理并行调度问题转化为多副本检索优化问题，应用最大流方法选择副本。3.3.1 节会对单表查询的并行调度方法进行详细的阐述。在多表查询的过程中，因为有连接操作，所以无法将多副本条件下的使用 MPP 计算框架的查询处理并行调度问题转化为多副本检索优化问题。因此，针对多表查询，本文提出了新的并行调度方法，3.3.2 节会对该方法进行详细的阐述。

3.3.1 适用于单表查询的调度方法

算法 3.1 的第 2~8 行描述了适用于单表查询的调度方法。首先，根据数据表的数据分布构造流网络，如行 3 所示。算法 3.2 会对构造流网络的过程进行详细描述。然后，因为 PushRelabelBinary 算法[58]是目前已知的解决多副本检索优化问题最好的算法，所以借鉴该算法获得副本选择结果，如行 4 所示。算法 3.3 会对副本选择的过程进行详细描述。接着，根据副本选择的结果确定并行调度，如行

5 所示。算法 3.5 会对该过程进行详细描述。接着，因为调度已经确定，所以更新磁盘负载信息如行 6 所示。最后返回调度结果，如行 7 所示。

算法 3.2 描述了根据数据表的数据分布构造流网络的过程。第一，获得数据分布矩阵 $M_{t\times n}$，如行 1 所示。其中 t 表示数据表的数据文件被分成的数据块数目，n 表示数据节点的数目。M_{ij} 的取值范围为{0, 1}。0 表示第 j 个磁盘上没有第 i 个数据块的副本，1 表示第 j 个磁盘上有第 i 个数据块的副本。第二，创建两个特殊的顶点，源点 v_s 和汇点 v_e，如行 2~3 所示。然后为每个数据块创建顶点 v_{bi}，如行 4~6 所示。其中 i 表示数据块的编号，为每个磁盘创建顶点 v_{dj}，其中 j 表示磁盘的编号，如行 7~9 所示。第三，创建从 v_s 到所有 v_{bi} 的边，其容量为 1，如行 10~12 所示。第四，若第 i 个数据块在第 j 个磁盘上有副本，则创建从 v_{bi} 到 v_{dj} 的边且容量为 1，如行 13~19 所示。第五，创建从 v_{dj} 到 v_e 的边且其容量为 1，如行 20~22 所示。从 v_{dj} 到 v_e 的边的容量表示第 j 个磁盘能提供的数据块的数目的上限。

算法 3.2：ConstructGraph

```
    输入：D
    输出：G
1   M_{t×n} ← DataLayout (D);
2   CreateVertex ( v_s );
3   CreateVertex ( v_e );
4   for i = 1 to t   do
5       CreateVertex (v_bi);
6   end
7   for j = 1 to n   do
8       CreateVertex (v_dj);
9   end
10  for i = 1 to t do
11      CreateEdge ( v_s , v_bi, 1);
12  end
13  for i = 1 to t do
14      for j = 1 to n do
15          if   M_ij == 1then
16              CreateEdge (v_bi, v_dj ,1);
17          end
18      end
19  end
20  for j = 1 to n
21     CreateEdge (v_dj, v_e ,1);
22  end
```

因为算法 3.2 需要构建以及遍历数据分布矩阵 $M_{t\times n}$，所以其时间复杂度为 O($t\times n$)。例如，某次查询涉及一张数据库表且其数据文件被分为数据块 0 和数据块 1，数据块 0 在磁盘 d_1、d_2 和 d_3 上有副本；数据块 1 在磁盘 d_2、d_3 和 d_4 上有副本，则构造出的流网络如图 3.3 所示：

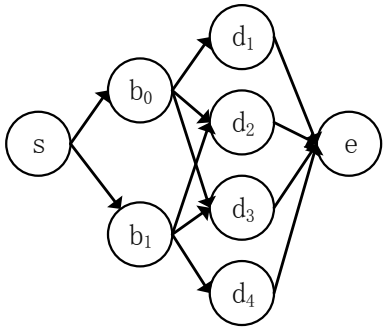

图 3.3 流网络

图 3.3 中的所有边的容量均为 1。

算法 3.3 描述了借鉴 PushRelabelBinary 算法完成副本选择的过程。

首先，找到初始的包含最优响应时间的区间(t_{min}, t_{max}]，如行 1~14 所示。假设所有的数据块从最慢的磁盘上读取可得到 t_{max}，假设所有的磁盘都和最快的磁盘速度一样，任务量完全平均分配，可得到 t_{min}。MaxTime(*j*)表示当由磁盘 *j* 获得所有的数据块时的查询响应时间，即，

$$(t\times byte)/diskSpeed[j]+(t\times byte)/netSpeed+L[j] \qquad \text{公式（3.5）}$$

假设有 *n* 个磁盘和 *t* 个数据块，若每个磁盘上选择的数据块的数目相同，则每个磁盘上选择的数据块的数目为$\lceil t/n \rceil$。MinTime(*j*)表示当由磁盘 *j* 获得均分的数据块时的查询响应时间，即，

$$(\lceil t/n \rceil\times byte)/diskSpeed[j]+(\lceil t/n \rceil\times byte)/netSpeed+L[j] \qquad \text{公式（3.6）}$$

为了保证响应时间为 t_{min} 的副本选择策略不存在，所有 MinTime(*j*)中的最小值减去 *minTimeSingleBlock* 得到初始的 t_{min}，如行 14 所示。

然后，用二分法搜索略小于最优响应时间的时间，如行 15~32 所示。在搜索过程中，第一步计算 t_{mid}，如行 16 所示。第二步初始化流网络，如行 17 所示。其包括源点的高度初始化为流网络中顶点的数目，流网络中其他顶点的高度初始

化为 0 以保证不存在从其他顶点到源点的 push 操作，从源点到所有代表数据块的顶点的流初始化为 1。第三步计算当响应时间为 t_{mid} 时，所有从磁盘顶点到汇点的边的容量，如行 18~20 所示。其中 CalculateCapacity (*t, j*) 计算当响应时间为 *t* 时，从 v_{dj} 到 v_e 的边的容量，如公式（3.7）所示：

算法 3.3: SRPushRelabelBinary

输入：*G, byte, L, diskSpeed, netSpeed, t, n*
输出：*R*
// *minTimeSingleBlock* 表示从最快的磁盘读取一个数据块所需要的时间
1 *minTimeSingleBlock* ← *MAXDOUBLE*;
2 $t_{\min} \leftarrow MAXINT$;
3 $t_{\max} \leftarrow 0$;
4 **for** $j = 1$ **to** *n* **do**
5 **if** MaxTime (*j*) > $t_{\max}$ **then**
6 $t_{\max} \leftarrow$ MaxTime (*j*);
7 **end**
8 **if** MinTime (*j*) < $t_{\min}$ **then**
9 $t_{\min} \leftarrow$ MinTime (*j*);
10 **end**
11 **if** *byte / diskSpeed* [*j*] < *minTimeSingleBlock* **then**
12 *minTimeSingleBlock* ← *byte / diskSpeed* [*j*];
13 **end**
14 $t_{\min} \leftarrow t_{\min} -$ *minTimeSingleBlock*;
15 **while** $(t_{\max} - t_{\min}) \geq$ *minTimeSingleBlock* **do**
16 $t_{mid} \leftarrow t_{\min} + (t_{\max} - t_{\min})/2$;
17 InitializePreFlow ();
18 **for** $j = 1$ to *n* **do**
// *edge* (*u, w*)表示从 *u* 到 *w* 的边
// *caps* [*e*]表示边 *e* 的容量
19 *caps* [*edge* (v_{dj}, v_e)] ← CalculateCapacity (t_{mid} , *j*);
20 **end**
21 apply push relabel operation
22 **if** *excess* [v_e] ≠ $|V'|$ **do**
23 $t_{\min} \leftarrow t_{mid}$;
24 **end**
25 **else**
26 $t_{\max} \leftarrow t_{mid}$;
27 **end**
28 **end**

```
29 for j = 1 to n do
30     caps [edge (v_dj, v_e )] ← CalculateCapacity ( t_min , j );
31 end
32 InitializePreFlow();
33 return PushRelabelIncremental (G, L,  t_min , n, byte, diskSpeed, netSpeed);
```

$$\frac{(t-L[j])\times diskSpeed[j]\times netSpeed}{(diskSpeed[j]+netSpeed)\times byte} \qquad \text{公式（3.7）}$$

从 v_{dj} 到 v_e 的边的容量用 *capacity* 表示，则磁盘 J 完成其上 I/O 任务所需时间为，

$$\frac{capacity\times byte}{diskSpeed}+\frac{capacity\times byte}{netSpeed}=t_{I/O} \qquad \text{公式（3.8）}$$

将公式（3.8）中的 $t_{I/O}$ 当作已知量，求解 *capacity*，可得公式（3.7）。第四步，执行压入重标记操作，如行 21 所示。第五步，如果汇点的溢出量等于数据块的总数目，则表示 t_{mid} 大于等于最优的响应时间，因此需在(t_{min}, t_{mid}]内寻找最优的响应时间，如果汇点的溢出量不等于数据块的总数目（汇点的溢出量不可能大于数据块的总数目），则表示 t_{mid} 小于等于最优的响应时间，因此需在(t_{mid}, t_{max}]内寻找最优的响应时间。当时间区间的上限和下限的差值小于 *minTimeSingleBlock* 时，t_{min} 略小于最优响应时间，如行 22~31 所示。

最后，计算当响应时间为 t_{min} 时磁盘顶点到汇点的容量，重新初始化流网络并调用 PushRelableIncremental 函数求最优副本选择策略，如行 29~33 所示。

因为 PushRelabelBinary 算法在最坏情况下的时间复杂度为 O(log|Q|*|Q|^3)，其中|Q|表示检索的 bucket 的数目，所以算法 3.3 在最坏情况下的时间复杂度为 O(log|t|*|t|^3)，其中|t|表示查询涉及的数据块的数目。

PushRelableIncremental 函数的伪代码如算法 3.4 所示：

算法 3.4 的处理流程如下所示：

1) 判断汇点的溢出量是否等于数据块的总数目，若等于，则表示已找到最优副本选择策略，结束；否则，进入下一步；
2) 选择负载最轻的磁盘并增加代表该磁盘的顶点到汇点的边的容量；
3) 执行压入重标记算法，转步骤 1）。

算法 3.4: PushRelabelIncremental

输入： $G, byte, L, diskSpeed, netSpeed, t_{\min}, n$

输出： R

```
1  while excess[v_e] ≠ |V'| do
2      IncrementMinCost(G, byte, L, diskSpeed,
                        netSpeed, t_min, n);
3      apply push relabel operation
4  end
5  return GetReplicationFromGraph(G);
```

图 3.4 展示了对图 3.3 应用算法 3.3 所得结果。其中，虚线表示流。由图 3.4 可知，选择数据块 b_0 在磁盘 d_3 上的副本，选择数据块 b_1 在磁盘 d_4 上的副本。

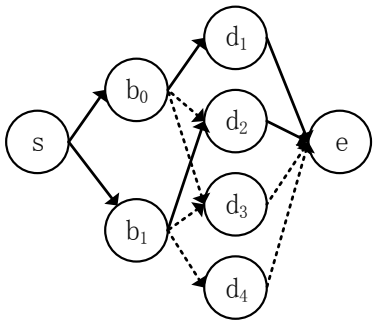

图 3.4 流

因为读磁盘操作在各个磁盘上并行执行，所以执行读磁盘操作所需时间由负载最重的磁盘决定。因此，每次均选择负载最轻的磁盘并增加代表该磁盘的顶点到汇点的边的容量可找到完成读磁盘操作所需最短时间。

算法 3.5 描述了根据副本选择的结果确定并行调度的过程。从根结点开始遍历查询计划树，为每个操作 v 确定执行节点 *v.execHosts*。为了减少通信开销，非叶子结点代表的操作的执行节点和其左孩子结点代表的操作的执行节点相同，如行 4 所示。叶子结点代表读磁盘操作。针对每个读磁盘操作，若副本所在节点可以作为执行节点，则选该节点为读磁盘操作的执行节点，否则，从所有执行节点 *allExecHosts* 中选择已分配任务最少的执行节点，如行 10~17 所示。依据任务量而不是磁盘负载选择执行节点的原因是远程读取数据并不影响执行节点的磁盘的负载。

算法 3.5: ExecutionNodeSelection

输入： T , R

输出： S

```
1   v ← root (T );
2   if  v_l ≠ NULL then
3       ExecutionNodeSelection ( v_l , R );
        // v.execHosts 表示执行操作 v 的节点集合
4       v.execHosts ← v_l.execHosts;
5   end
6   else if v_r ≠ NULL then
7       ExecutionNodeSelection ( v_r , R );
8   end
    // allExecHosts 表示执行节点的全集
9   allExecHosts ← GetExecHosts();
    // v.dataHosts 表示含有操作 v 所需数据的节点集合
10  for dataHost ∈ v.dataHosts do
11      if dataHost ∈ allExecHosts then
12          add dataHost to v.execHosts
13      end
14      else
15          add LeastLoad (allExecHosts) to v.execHosts
16      end
17  end
```

3.3.2 适用于多表查询的调度方法

首先按原 Impala2.0 的调度方法进行调度并计算查询处理的代价，如算法 3.1 的行 9~11 所示。因为 Impala 使用哈希连接算法，将右表的数据发送到左表所在的各个节点上。所以左表的读磁盘操作的并行度决定了连接操作的并行度。因此，左表的读磁盘操作是关键操作，其并行度应该最大化。相反的，右表的读磁盘操作是非关键操作。非关键操作的执行节点分为和关键操作的执行节点相同或不同两种。若相同，则没有通信代价，但并行性下降。若不同，则有通信代价，但并行性提升。因此，必须结合 Impala 查询处理代价模型，搜索近似最优解，如算法 3.1 的行 12~23 所示。接着更新执行读磁盘操作的磁盘的负载信息如算法 3.1 的行 24 所示。最后，返回调度结果，如算法 3.1 的行 25 所示。在搜寻代价尽可能小

的调度的过程中，先为关键操作选择副本并记录执行关键操作的节点集合，如行 12 所示。算法 3.6 会对该过程进行详细描述。然后搜索 *maxTime* 次调度策略，每次只变化非关键操作的副本选择，如行 13~23 所示。算法 3.7 会对每次为非关键操作选择副本和执行节点的过程进行详细描述。

算法 3.6 描述了为关键操作选择副本并记录执行关键操作的节点集合的过程。其和 Impala2.0 原有的调度算法的不同之处在于：第一，不是每次选择已分配任务最少的磁盘，而是选择负载最轻的磁盘，如行 7 所示。第二，更新该磁盘负载以反映该磁盘 I/O 任务增加。第三，记录选择的执行节点集合 I。为非关键操作选择副本以及为非叶子结点选择执行节点的过程均依赖于 I。

算法 3.6：SelectReplicationImportant

输入：D，L

输出：R

```
1   I ← Φ;
2   R_i ← Φ;
3   node ← GetImportantNode (D);
4   scanRanges ← GetScanRanges (node);
5   for b ∈ scanRanges do
        // N_b 表示副本所在节点集合
6       N_b ← GetLocations (b);
7       n_k, d_j ← LeastLoad (N_b, L);
8       r ← GetReplication (b, d_j);
9       add n_k to I
10      add r to R_i
11      UpdateLoadInfo (d_j);
12  end
13  return R_i, I;
```

因为算法 3.6 的最外层循环的次数为关键操作涉及的数据表所包含的 Scan Range 的数目 *numOfIScanRange*，又因为一个 Scan Range 的副本数为常数，所以算法 3.6 的时间复杂度为 O(*numOfIScanRange*)。

算法 3.7 描述了为非关键操作选择副本的过程。为非关键操作选择副本的过程中，需综合考虑并行度和通信代价，若数据块 b 的副本所在节点集 N_b 中没有关键操作的副本，即 $N_b \cap I \neq \Phi$，则表示无论为该数据块选择哪个节点上的副本，

均存在通信代价，此时只需考虑均衡磁盘负载即可，否则，随着 *time* 不断增大，随机数对 *maxTime* 取余的结果小于 *time* 的概率越来越大，副本的节点选择由 I 中的节点逐渐向其它节点扩散，如算法 3.7 的行 7~17 所示。

因为算法 3.7 是搜索并行调度策略，并没有真正分配任务，所以，不同于算法 3.1，算法 3.7 的磁盘负载增加是临时的。在下一次搜索并行调度策略时，磁盘负载重置为未分配任务前的状态。

因为多表查询处理并行调度方法的搜索空间包含 Impala2.0 原有的调度方案，所以在代价模型完全准确的前提下，为多表查询处理并行调度方法的查询响应时间一定小于等于大数据实时查询系统 Impala2.0 的查询响应时间。

算法 3.7： SelectReplicationUnimportant

输入： D，I，L' *time, maxTime*

输出： R_u

```
1  R_u ← Φ;
2  nodes ← GetUnimportantNode (D);
3  for each node∈nodes do
4      scanRanges ← GetScanRanges (node);
5      for b ∈ scanRanges do
6          N_b ← GetLocations (b);
7          if N_b ⋂ I ≠ Φ && rand() % maxTime < time then
8              N_c ← N_b ⋂ I;
9              d_j ← LeastLoad (N_c, L');
10         end
11         else
12             d_j ← LeastLoad (N_b, L');
13         end
14         r ← GetReplication (b, d_j);
15         add r to R_u
16     end
17 end
18 return R_u;
```

因为算法 3.7 遍历所有非关键操作所涉及的数据表的 Scan Range，又因为一个 Scan Range 的副本数为常数，所以算法 3.7 的时间复杂度为

O(*numOfUScanRange*)。其中 *numOfUScanRange* 表示所有非关键操作所涉及的数据表的 Scan Range 的数目。

3.4 调度方法对副本一致性的影响

HDFS 负责管理副本的一致性。一方面，大数据实时查询中的插入、删除和更新操作的频率较低，另一方面，与大数据实时查询系统 Impala2.0 原有的调度方法相比，基于副本选择的大数据实时查询处理并行调度方法并未改变副本数，因此，一致性控制的代价保持不变。

3.5 本章小结

本章首先对本课题提出的基于副本选择的大数据实时查询处理并行调度方法进行了一般性的概述，其将所有查询分为单表查询和多表查询两类，然后分别确定并行调度。其次详细介绍了基于副本选择的并行调度方法使用到的一些概念及定义，为后续的章节做铺垫；接着详细介绍适用于单表查询的并行调度方法和适用于多表查询的并行调度方法，并对其中比较重要的算法分析了时间复杂度；最后分析了基于副本选择的大数据实时查询处理并行调度方法对副本一致性控制的影响。

第4章 Impala 查询处理代价模型

4.1 概述

代价模型是并行调度方法的重要组成部分。由上一章可知，若是多表查询，则基于副本选择的大数据实时查询处理并行调度方法需结合提出的 Impala 查询处理代价模型搜索近似最优调度。4.2 节将详细介绍本课题对 Impala 查询处理中常见操作建模，其是 Impala 查询处理代价模型的基础。4.3 节将详细介绍 Impala 查询处理代价模型。为提高 Impala 查询处理代价模型的准确度，本课题估计查询处理的中间结果，4.4 节将给出详细介绍。

4.2 操作模型

因为 Impala 查询处理中的聚集操作较复杂，所以本节未对聚集操作构建模型，本节主要对 Impala 查询处理中的读磁盘操作、选择操作、传输操作和连接操作构建模型。

读磁盘操作主要包括三个步骤，第一，磁臂移动到数据所在磁道；第二，磁头旋转到 I/O 请求所请求的起始数据块位置；第三，读取数据。本课题假设若 I/O 任务 *task* 分配给磁盘 *j*，且磁盘 *j* 在分配 I/O 任务 *task* 之前的负载为 *L*，则磁盘 *j* 在 L_{ms} 后开始执行 I/O 任务 *task*，一旦开始执行 I/O 任务 *task*，不会执行其他 I/O 任务。

选择操作主要包括两个步骤，第一，以其孩子结点代表的操作的输出元组集合为输入元组集合；第二，从输入元组集合逐一取出元组并判断是否满足条件，若满足，则将该元组拷贝到输出元组集合。

传输操作就是将数据从集群的一个节点传输到集群的另一个节点。

Impala 使用哈希连接，其以参与连接操作的右表为 build 表，以参与连接操作的左表为 probe 表。连接操作的步骤如下：

1) 为参与连接操作的左表的每个元组求哈希值；

2) 为参与连接操作的右表的每个元组求哈希值；

3) 为右表在内存中构建哈希表；

4) 左表作为 probe 表，若找到匹配的一对，则创建一个新的元组，并将左表的元组和右表的元组的内容拷贝到新的元组中。

因为在 Impala 系统中数据文件已经被分成数据块，所以读磁盘操作、选择操作、转移操作和连接操作会在多个节点上并行执行，每个节点处理一部分数据。

4.3 代价模型

在详细介绍 Impala 查询处理代价模型之前，先给出 Impala 查询处理的代价的定义。

定义 4.1 Impala 查询处理的代价。Impala 查询处理的代价 C 为从查询开始处理时刻 t_{start} 到预估所有连接操作完成时刻 $ct_{allJoin}$ 的时间间隔，即，

$$C = ct_{allJoin} - t_{start} \quad \text{公式（4.1）}$$

由上一节知，该本课题未对聚集操作构建模型。因此，本课题定义的 Impala 查询处理的代价不包括聚集操作的代价。$ct_{allJoin}$ 等于查询计划树的最顶端的 Join 结点所代表的操作完成的时刻 $ct_{topJoin}$。因为连接操作在多个节点并行执行，所以最顶端的Join结点所代表的操作的完成时刻由执行该操作的节点集合中最慢的一个决定的，即，

$$ct_{topJoin} = \max_{k \in N_{topJoin}} (ft_{topJoin}[k]) \quad \text{公式（4.2）}$$

其中，$N_{topJoin}$ 表示执行最顶端的 Join 结点所代表的操作的节点集合。$ft_{topJoin}[k]$表示 n_k 完成最顶端的 Join 结点所代表的操作的时刻。节点 n_k 完成连接操作的时刻 $ft_{join}[k]$等于其开始连接操作的时刻 $bt_{join}[k]$加上处理连接操作的时间 $t_{join}[k]$，即，

$$ft_{join}[k] = bt_{join}[k] + t_{join}[k] \quad \text{公式（4.3）}$$

由上一节的连接操作模型知，节点 n_k 处理连接操作的时间 $t_{join}[k]$包括节点 n_k 根据连接的属性对其上参与连接操作的所有元组求哈希值的时间 $t_{hashTuple}[k]$，将参与连接操作的所有元组插入哈希表的时间 $t_{insertTuple}[k]$，为右表在内存建立哈希索引的时间 $t_{build}[k]$，将左表的元组和右表的元组连接的时间 $t_{joinTuple}[k]$，即

$$t_{join}[k] = t_{hashTuple}[k] + t_{insertTuple}[k] + t_{build}[k] + t_{joinTuple}[k] \quad \text{公式（4.4）}$$

因为开始查询处理之前右表建立的哈希表的大小较难估计，所以未给出右表在内存建立哈希索引的时间的计算公式，将其作为查询的一个参数。节点 n_k 根据连接的属性对其上参与连接操作的所有元组求哈希值的时间取决于对一个元组求哈希值的时间 t_{hash}（其为参数）和节点 n_k 上待操作的元组的个数 $numOfTuples[k]$，即，

$$t_{hashTuple}[k] = t_{hash} \times numOfTuples[k] \quad \text{公式（4.5）}$$

节点 n_k 将参与连接操作的所有元组插入哈希表的时间取决于将一个元组插入哈希表的时间 t_{insert}（其为参数）和节点 n_k 上待操作的元组的个数 $numOfTuples[k]$，即，

$$t_{insertTuple}[k] = t_{insert} \times numOfTuples[k] \quad \text{公式（4.6）}$$

两个元组连接的过程分为两步，第一步创建一个空的元组，第二步将两个元组的内容拷贝到新创建的元组。因此，节点 n_k 上左表的元组和右表的元组连接的时间可由公式（4.7）计算，

$$t_{joinTuple}[k] = (t_{newTuple} + t_{copyLeft} + t_{copyRight}) \times numOfPairs[k] \quad \text{公式（4.7）}$$

其中，$t_{newTuple}$ 表示创建一个新的元组的时间（其为参数），$t_{copyLeft}$ 表示将左表的一个元组的内容拷贝到新创建的元组的时间（其为参数），$t_{copyRight}$ 表示将右表的一个元组的内容拷贝到新创建的元组的时间（其为参数），$numOfPairs[k]$表示节点 n_k 上匹配的元组对的数目。

只有当节点 n_k 上的左表的数据和右表的数据均准备好时，节点 n_k 上的 Impala 服务进程才开始处理连接操作。因此，节点 n_k 上的连接操作的开始执行时刻取决于左表的数据就绪的时刻和右表的数据就绪时刻的最大值。其中左表的数据就绪的时刻也就是节点 n_k 完成处理 Join 结点的左孩子结点所代表的操作的时刻 $ft_{lop}[k]$，右表的数据就绪的时刻也就是节点 n_k 完成处理 Join 结点的右孩子结点所代表的操作的时刻 $ft_{rop}[k]$，即，

$$bt_{join}[k] = \max(ft_{lop}[k], ft_{rop}[k]) \quad \text{公式（4.9）}$$

因为 Impala 系统中的查询计划树是左深结构的，所以 Join 结点的左孩子可能是 Scan 结点、Select 结点和 Join 结点，Join 结点的的右孩子结点一定是 Exchange 结点。Exchange 结点代表传输操作。处理 Exchange 结点的孩子结点代表的操作的节点是传输的源端。处理 Exchange 结点的左兄弟结点代表的操作的节点是传输的目的端。因为 Exchange 的孩子结点所代表的操作可能在多个节点上并行执行，所以传输到节点 n_k 的操作完成时刻 $ft_{exchange}[k]$ 由最后完成的决定，即，

$$ft_{exchange}[k]=\max_{k'\in N_k}(bt_{exchange}[k']+tran_k[k']) \qquad \text{公式（4.10）}$$

其中，N_k 表示所有需要将数据发送到节点 n_k 的节点集合。$tran_k[k']$ 表示将数据从节点 $n_{k'}$ 传输到节点 n_k 所需时间。由上一节的传输操作模型知，其由需要传输的字节量 $TransferByte$ 和网络带宽 Net_{band}（单位为 Mbps）决定（其为参数），即

$$tran_k[k']=TransferByte\times 8/(Net_{band}\times 10^6) \qquad \text{公式（4.11）}$$

Exchange 结点只有一个孩子结点，该孩子结点可能是 Scan 结点，也可能是 Select 结点。因此，代表从从节点 $n_{k'}$ 传输到节点 n_k 的操作开始的时刻 $bt_{exchange}[k']$ 为节点 $n_{k'}$ 上的选择操作或读磁盘操作完成的时刻 $ft_{selectOrScan}[k']$，即，

$$bt_{exchange}[k']=ft_{selectOrscan}[k'] \qquad \text{公式（4.12）}$$

节点 $n_{k'}$ 上的选择操作的结束时刻由其上选择操作的开始时刻和选择操作的执行时间决定，即，

$$ft_{select}[k']=bt_{select}[k']+t_{select}[k'] \qquad \text{公式（4.13）}$$

由上一节的选择操作模型知，节点 $n_{k'}$ 上的选择操作的执行时间包括判断输入元组是否满足条件的时间 $t_{judgeTuple}[k']$ 和构造输出元组集合的时间 $t_{constructOutputTuples}[k']$，即，

$$t_{select}[k']=t_{constuctOutputTuples}[k']+t_{judgeTuple}[k'] \qquad \text{公式（4.14）}$$

其中节点 $n_{k'}$ 上的构造输出元组集合的时间取决于将一个元组拷贝到输出集合的时间 t_{copy}（其为参数）和待拷贝的元组数 $numOfCopiedTuples[k']$，即，

$$t_{constuctOutputTuples}[k']=t_{copy}\times numOfCopiedTuples[k'] \qquad \text{公式（4.15）}$$

其中待拷贝的元组数 *numOfCopiedTuples*[k']属于中间结果，4.4 节将详细介绍如何估计中间结果。当有多个条件时，最少需要判断一个条件。例如，当多个条件相与时，如果第一个条件不满足，后面的条件也就无需判断了。平均需要判断一半的条件。因此，判断节点 $n_{k'}$上输入元组是否满足条件的时间 $t_{judgeTuple}[k']$ 可由公式（4.16）计算，

$$t_{judgeTuple}[k'] = (1 + (numOfConditions - 1)/2) \times t_{judge}) \times numOfTuples[k'] \quad \text{公式（4.16）}$$

其中，*numOfConditions* 表示条件的数目，t_{judge} 表示判断一个条件所需时间（其为参数），*numOfTuples*[*k*]表示节点 n_k 上待操作的元组的个数。选择操作的开始时刻等于其孩子结点所代表的操作的结束时刻，Select 结点的孩子结点一定是 Scan 结点，因此，

$$bt_{select}[k'] = ft_{scan}[k'] \quad \text{公式（4.17）}$$

因为节点 $n_{k'}$可能包含多个磁盘 $DISK_{k'}$，所以节点 $n_{k'}$上的读磁盘操作完成的时刻取决于最慢的磁盘，即，

$$ft_{scan}[k'] = \max_{j \in DISK_{k'}} (bt_{scan}[j] + t_{scan}[j]) \quad \text{公式（4.18）}$$

其中，$t_{scan}[j]$表示 Impala 服务进程执行从磁盘 *j* 读取和本次查询相关的数据所需时间，$bt_{scan}[j]$表示 Impala 服务进程开始从磁盘 *j* 读取和本次查询相关的数据的时刻。由定义 3.7 知，$bt_{scan}[j]$取决于磁盘 *j* 的负载 *L*[*j*]和当前时刻 t_{now}，即，

$$bt_{scan}[j] = L[j] + t_{now} \quad \text{公式（4.19）}$$

由上一节的读磁盘操作模型知，Impala 服务进程执行从磁盘 *j* 扫描和本次查询相关的数据所需时间由磁盘 j 的平均寻道时间 $t_{seek}[j]$（其为参数），延迟时间 $t_{latency}[j]$（其为参数）和读取操作所需时间 $t_{read}[j]$（其为参数）组成，即，

$$t_{scan}[j] = t_{seek}[j] + t_{latency}[j] + t_{read}[j] \quad \text{公式（4.20）}$$

代价模型中的参数的取值主要与集群环境相关，即，Impala 服务进程所在节点的 CPU 主频，Impala 服务进程所在节点的内存大小，Impala 服务进程所在节点的磁盘性能和集群的网络带宽。

4.4 中间结果估计

中间结果估计有利于提高代价模型的准确性。例如，用户提交查询语句“SELECT * FROM A WHERE A.t > 10”，公式（4.15）中的待拷贝的元组数 *numOfCopiedTuples*[*k*']是 A 表中 t 属性值大于 10 的元组的数目。该值只有在查询执行的过程中才能得到。因此，公式（4.15）中的待拷贝的元组数是中间结果并且无法通过 2.1.3 节中的统计信息提供较准确的中间结果估计。由 1.2.2.1 节知，抽样方法、参数化方法和直方图方法均能提供较准确的中间结果估计。其中，直方图方法具有运行时开销小，无需假设数学分布，通过耗用少量的空间便可达到较高的估算准确性的优点。而在直方图方法中，Maxdiff(V, A)直方图误差率较低。因此，本课题采用 Maxdiff(V, A)直方图。Maxdiff(V, A)直方图考虑相邻值之间的区域。若桶的数目为 *x*，则 Maxdiff(V, A)直方图首先按值对属性进行排序，然后选择 *x* 个区域值最大的桶。

本课题采用的选择度计算方法具体如下：相等的选择度可由相异值的倒数得到。不等、小于、小于等于、大于等于和大于等二元谓词的选择度根据 Maxdiff(V, A)直方图计算得到。

若用 selectivity(p_1)表示谓词 p_1 的选择度，用 selectivity(p_2)表示谓词 p_2 的选择度，则谓词 p_1 和谓词 p_2 相与的选择度，即 selectivity(p_1 and p_2)可由公式（2.1）计算。谓词 p_1 和谓词 p_2 相或的选择度，即 selectivity(p_1 or p_2)可由公式（4.21）计算：

$$\text{selectivity}(p_1 \text{ or } p_2) = \text{selectivity}(p_1) + \text{selectivity}(p_2) - \text{selectivity}(p_1 \text{ and } p_2) \quad \text{公式（4.21）}$$

不满足谓词 p_1 的选择度，即 selectivity(not p_1)可由公式（2.3）计算，属性 *a* 介于 n_1 和 n_2 之间的选择度，即 selectivity(*a* between n_1 and n_2)可由公式（2.4）计算，属性 *a* 不介于 n_1 和 n_2 之间的选择度，即 selectivity(*a* not between n_1 and n_2)可由公式（2.5）计算。

因为一个节点的某个操作可能只处理数据库表的一部分数据，所以该操作的选择度可能和整个表的选择度不相同。例如，用户提交查询语句“SELECT * FROM A WHERE A.t > 10”。A 表的数据量为 128MB，HDFS Block 的大小为 64MB。A

表的数据存储了数据块 b_{A1} 和 b_{A2} 中。数据块 b_{A1} 中的 A.t 的值均为 5，存储在节点 n_1 的磁盘上，数据块 b_{A2} 中的 A.t 的值均为 12，存储在节点 n_2 的磁盘上，则节点 n_1 上的选择操作的选择度为 0，节点 n_2 上的选择操作的选择度为 1。因为数据块的划分是物理划分，按数据块收集统计信息较困难，所以本课题假设对于同一条件而言，同一个表的各个数据块的选择度相同。

4.5 本章小结

这一章介绍了本课题提出的 Impala 查询处理代价模型。首先对 Impala 查询处理过程中的操作构建模型，然后给出了 Impala 查询处理代价的定义，接着详细介绍了代价计算的公式以及代价计算公式的依据，最后介绍了中间结果估计相关的统计信息收集和选择度计算。

第5章 系统实现

5.1 系统架构

本文的第 3 章详细介绍了基于副本选择的并行调度方法，第 4 章详细介绍了 Impala 查询处理代价模型。为了验证本文提出的调度方法的有效性及高效性，本课题将上述研究工作集成到大数据实时查询系统 Impala2.0，集成后的系统架构如图 5.1 所示：

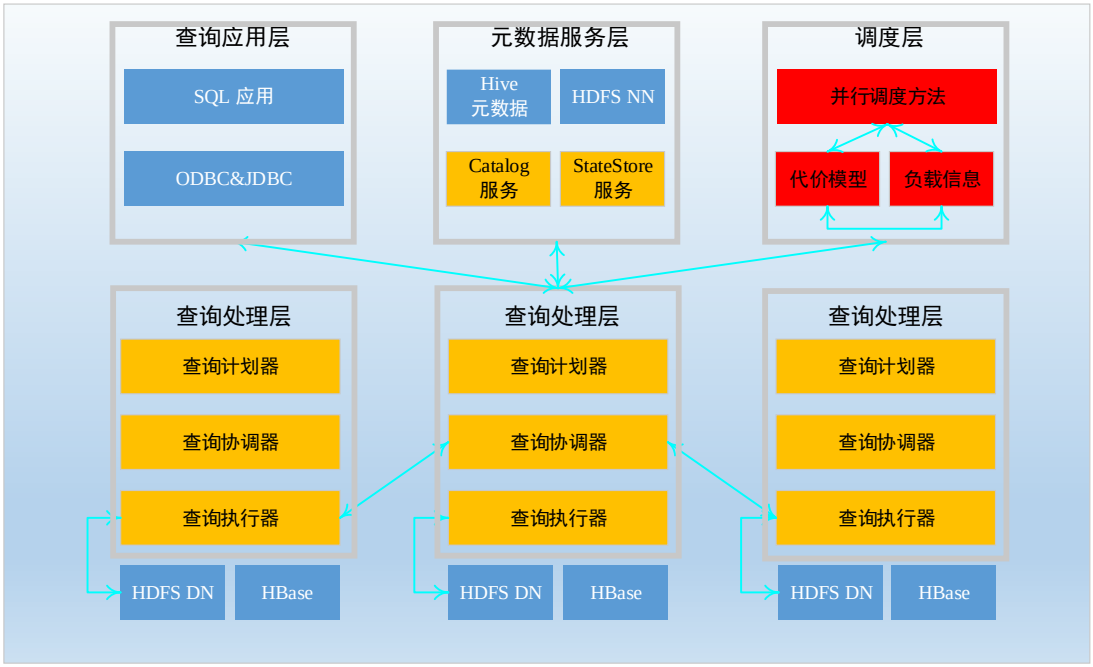

图 5.1 集成后的 Impala 2.0 体系架构

由图 5.1 可知，与集成前的 Impala 2.0 体系架构相比，集成后的 Impala 2.0 体系架构主要修改了调度层，其中红色矩形框表示新增加的模块，包括并行调度方法模块、代价模型模块和负载信息模块。5.2 节将详细介绍各个模块的实现。

5.2 模块实现

5.2.1 并行调度方法模块

由 2.1.2 节可知 Impala2.0 提供纯虚类 Scheduler 以规范调度模块的接口。因

此，本课题通过 Scheduler 的另一个派生类 ImprovedScheduler 实现提出的基于副本选择的并行调度方法。其主要接口为 ComputeScanRangeAssignment(TQueryExecRequest, QuerySchedule) ImprovedScheduler 类根据查询执行请求 TQueryExecRequest，构造查询处理并行调度 QuerySchedule。Impala2.0 在执行环境类 ExecEnv 的构造函数中完成调度器的设置。ExecEnv 类包含成员变量 boost::scoped_ptr<Scheduler> scheduler_，其为指向调度器的智能指针，只需让该指针指向 ImprovedScheduler，就可完成调度方法的替换。

若选择的副本所在节点没有 Impala 服务进程，则需为该副本选择其他节点上的 Impala 服务进程远程读取数据。相比于本地读取，远程读取的代价更高。因此，需避免远程读取。但另一方面，若有 Impala 服务进程的节点的磁盘负载已经较高，则应选择没有 Impala 服务进程但负载较轻的节点的磁盘。由 2.1.2 节知，通过初始化没有 Impala 服务进程的节点的已分配字节量为 2^{63}-1B，而有 Impala 服务进程的节点的已分配字节量被初始化为 0B，SimpleScheduler 就可实现在副本选择时优先考虑有 Impala 服务进程的节点的目的。类似的，若磁盘所在的节点没有 Impala 服务进程，则 ImprovedScheduler 在该磁盘的负载量的真实值的基础上再增加 *initialLoad*（单位是 s），若磁盘所在的节点有有 Impala 服务进程，则该磁盘的负载量就是真实值。

5.2.2 代价模型模块

本课题通过 ParallelCostModel 类实现 Impala 查询处理代价模型。其主要接口为 Cost Calculate(QuerySchedule, PerHostLoad)。ParallelCostModel 类根据查询处理并行调度 QuerySchedule 和集群负载信息 PerHostLoad，按照 4.3 节介绍的 Impala 查询处理并行代价模型返回预估的代价 Cost，其类型为 uint64_t。

5.2.3 负载信息模块

由 4.3 节可知 Impala 查询处理代价模型依赖于集群负载信息。本课题采用集中式的负载信息收集机制。每个节点均有负载信息汇报器，整个集群有一个负载信息收集器。首先，节点负载信息汇报器到集群负载信息收集器注册。然后，各

个节点上的负载信息汇报器按照预设的时间周期定期向负载信息收集器汇报相应节点的负载信息，由负载信息收集器形成集群负载信息。节点负载信息汇报器由 LoadReportor 类实现，而集群负载信息收集器由 LoadCollector 类实现。

5.2.4 统计信息模块

由 4.4 节可知 Impala 查询处理代价模型依赖于统计信息。本课题采用 Maxdiff(V, A)直方图方法，其由 Histogram 类实现，直方图的信息存储在 MySQL 数据库中，针对查询涉及的数据库表的列建立直方图，例如表 5.1 所示为某个数据库表的某列建立的 Maxdiff(V, A)直方图。

表 5.1 直方图

Cardinality	lowBound	highBound
50	102	106
70	109	112
40	120	123

由表 5.1 可知，该列的值从 102 到 106 有 50 个，从 109 到 112 有 70 个，从 120 到 123 有 40 个，该列的值共有 160 个。

5.3 本章小结

本章首先介绍了集成本课题提出的调度方法后的系统总体框架，然后详细介绍了新添加的调度层的并行调度方法模块、代价模型模块、负载信息模块和统计信息模块的实现。

第6章 实验评估

6.1 实验环境

本实验在 Hadoop 集群和 Impala 大数据实时查询系统上实现，Hadoop 集群由 13 个节点组成，其操作系统为 Ubuntu 12.04 LTS 64-bit，Hadoop 集群通过 Cloudera Manager 安装部署，版本号为 CDH5.2.0，另外，Hadoop 集群之上编译部署的 Impala 版本为 2.0，每个节点上运行的服务及硬件相关配置如表 6.1 所示：

表 6.1 集群环境

节点主机	运行的服务	硬件配置
Impala1	HDFS NameNode HDFS SecondaryNameNode Hive Metastore Server Hive Server2 MapReduce JobTracker	内存：32GB 磁盘空间：2TB 主频：1.6GHZ×4
Impala2	HDFS DataNode Hive Gateway Impala StateStore Impala Catalog Impala Daemon Collector Reportor MapReduce TaskTracker	内存：32GB 磁盘空间：2TB 主频：1.6GHZ×4
Impala3~ Impala9	HDFS DataNode Hive Gateway Impala Daemon Reportor MapReduce TaskTracker	内存：32GB 磁盘空间：2TB 主频：1.6GHZ×4
Impala10~ Impala12	Zookeeper	内存：32GB 磁盘空间：2TB 主频：1.6GHZ×4
Impala13	Cloudera Management Service	内存：32GB 磁盘空间：2TB 主频：1.6GHZ×4

其中，Collector 是负载信息收集服务，Reportor 是负载信息汇报服务。

6.2 实验设置

本实验包括参数调优和对比实验两大部分。其中，参数调优实验是对调度方

法涉及的参数进行调优，包括 *initialLoad* 参数和 *maxTime* 参数调优。对比实验包括无并发查询请求情况下的调度方法对比和有并发查询请求情况下的调度方法对比两部分，其中无并发查询请求情况下的调度方法又分为适用于单表查询的调度方法对比、适用于多表查询的调度方法对比和不同副本数的调度方法对比三部分，实验设置具体如下：

1) *initialLoad* 参数调优

由 5.2.1 节知，ImprovedScheduler 对于没有 Impala 服务进程的节点的所有磁盘，在其真实负载的基础上再增加 *initialLoad*。本实验的目的是寻找使查询响应时间较短的 *initialLoad* 取值。

2) *maxTime* 参数调优

由 3.3.2 节知，适用于多表查询的调度方法的搜索次数为 *maxTime* 次。本实验的目的是寻找使查询响应时间较短的 *maxTime* 取值。

3) 无并发查询请求情况下的适用于单表查询的调度方法对比实验

本实验对比 3.3.1 节中提出的适用于单表查询的调度方法和如 2.1.2 节所述的 Impala2.0 原有的调度方法，其目的是验证本文提出的适用于单表查询的调度方法的有效性及其对查询响应时间的影响。

4) 无并发查询请求情况下的适用于多表查询的调度方法对比实验

本实验对比 3.3.2 节中提出的适用于多表查询的调度方法和如 2.1.2 节所述的 Impala2.0 原有的调度方法，其目的是验证本文提出的适用于多表查询的调度方法的有效性及其对查询响应时间的影响。

5) 不同副本数的调度方法对比实验

本实验在不同副本数情况下，对比本课题提出的调度方法和如 2.1.2 节所述的 Impala2.0 原有的调度方法，其目的是验证在不同副本数情况下本文提出的调度方法的有效性。

6) 有并发查询请求情况下的调度方法对比实验

本实验在并发查询请求情况下，对比本课题提出的调度方法和如 2.1.2 节所述的 Impala2.0 原有的调度方法，其目的是验证在并发查询请求情况下本文提出

的调度方法的有效性。

6.3 实验数据

实验数据源由 TPC-DS[60]数据库提供，其由 25 个独立的基本表组成。本实验的数据集规模为 20GB、40GB、60GB 和 80GB。Cloudera 公司做了 Impala2.0 的性能测试实验[61]。其指出 TPC-DS 中的交互型查询语句包括 q19、q42、q52、q55、q63、q68、q73 和 q98。其中，在本实验的环境中可执行的查询语句包括 q19、q42、q52 和 q55。因此，本实验以上述 4 条查询语句为测试集，涉及 date_dim、store_sales、item、customer、customer_address 和 store 数据表。20GB、40GB、60GB 和 80GB 数据集规模下的上述 6 张数据库表名及表行数如表 6.2 所示：

表 6.2 不同数据集规模下数据库表及其总行数

	20GB	40GB	60GB	80GB
date_dim	7.30×10^4	7.30×10^4	7.30×10^4	7.30×10^4
store_sales	5.76×10^7	1.15×10^8	1.73×10^8	2.30×10^8
item	2.80×10^4	5.20×10^4	7.40×10^4	9.60×10^4
customer	2.66×10^5	6.00×10^5	9.33×10^5	1.27×10^6
customer_address	1.33×10^5	3.00×10^5	4.66×10^5	6.33×10^5
store	44	112	178	244

因为在本实验的环境中可执行的 TPC-DS 查询语句中不包含单表查询语句，因此本实验选取 store_sales 表，并以“select sum(ss_ext_sales_price) extended_price from store_sales”为单表测试语句。因为在所有数据库表中 store_sales 表数据量最大，即 HDFS Block 数目最多，又因为 store_sales 表与数据集规模的增长基本保持一致，即数据集规模增大一倍，store_sales 表的数据量也基本增大一倍，所以选取 store_sales 表完成单表测试可达到较好的实验效果。不同数据集规模下 store_sales 表的数据量及其 HDFS Block 的数目如表 6.3 所示：

表 6.3 不同数据集规模下 store_sales 表的数据量及其 HDFS Block 的数目

	20GB	40GB	60GB	80GB
数据量	5.61GB	11.35GB	17.19GB	23.07GB
HDFS Block 数目	45	91	138	185

6.4 实验结果与分析

6.4.1 *maxTime* 参数调优

由 3.3.2 节知，适用于多表查询的调度方法的搜索次数为 *maxTime*。在 TPC-DS 80GB 数据集规模下，每设置一个 *maxTime*，则执行 q19、q42、q52 和 q55 查询语句，记录查询响应时间，以观测 *maxTime* 参数对查询响应时间的影响。每个查询语句均执行多次得到平均响应时间，以 4 条查询语句的平均响应时间之和为总响应时间，实验结果如图 6.1 所示：

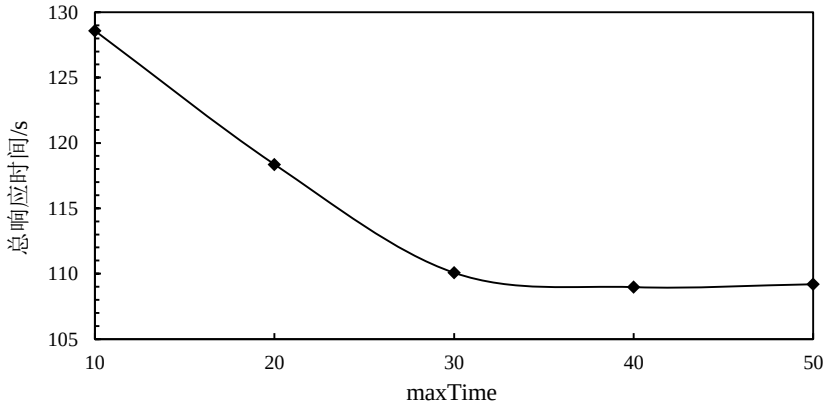

图 6.1 *maxTime* 对查询响应时间的影响

由图 6.1 可知，查询响应时间随 *maxTime* 的增加先减少后增大，当 *maxTime* 为 40 时，响应时间最小。其原因如下，随着 *maxTime* 的增加，搜索空间和调度方法本身所需时间同时增大。因为搜索空间增大，所以适用于多表查询的调度方法可找到使查询响应时间更少的调度策略。但是当搜索空间增大到一定程度时，调度方法本身所需时间的增加量会超过更优的调度策略造成的查询响应时间的减少量，造成总的查询响应时间增加。

6.4.2 *initialLoad* 参数调优

由 5.2.1 节知，ImprovedScheduler 对于没有 Impala 服务进程的节点的所有磁盘，在其真实负载的基础上再增加 *initialLoad*。在 TPC-DS 80GB 数据集规模下，关闭 Impala2 上的 Impala 服务进程，设置 *maxTime* 为 40，每设置一个 *initialLoad*，则执行 q19、q42、q52 和 q55 查询语句，以观测 *initialLoad* 参数对查询响应时间

的影响。每个查询语句均执行多次并求平均，以 4 条查询语句的响应时间之和为总响应时间，实验结果如图 6.2 所示：

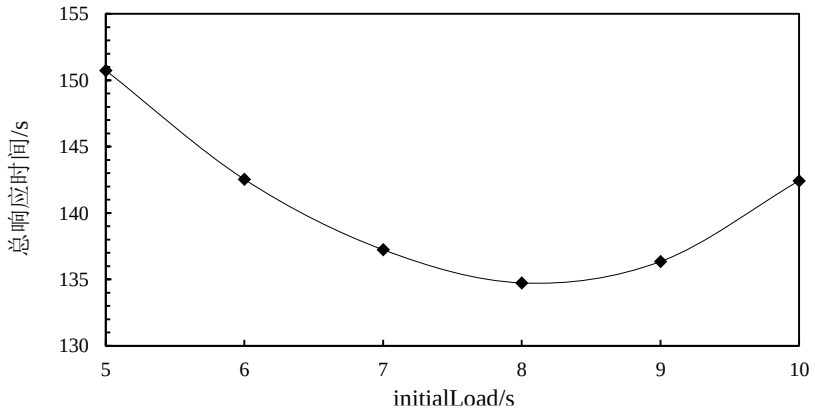

图 6.2 *initialLoad* 对查询响应时间的影响

由图6.2可知，查询响应时间随 *initialLoad* 的增加先减少后增大，当 *initialLoad* 为 8 时，响应时间最小。其原因可由下述例子说明。例如，数据块 *b* 在磁盘 d_0 和 d_1 上有副本。磁盘 d_0 所在的节点有 Impala 服务进程，磁盘 d_1 所在的节点没有 Impala 服务进程。当 *initialLoad* 值为 0 时，影响选择哪个磁盘上的副本的因素仅仅是两个磁盘的负载。若磁盘 d_0 和磁盘 d_1 的负载相同，则选择磁盘 d_0 上的副本和选择磁盘 d_1 上的副本的概率相同。但是，此时应该选择磁盘 d_0 上的副本。其原因是若选择磁盘 d_0 上的副本，则可由磁盘 d_0 所在的节点执行读磁盘操作，若选择磁盘 d_1 上的副本，则还需另外寻找有 Impala 服务进程的节点执行远程读取操作。因此，相比于选择磁盘 d_0 上的副本，选择磁盘 d_1 上的副本会造成响应时间延长。当 *initialLoad* 不为 0 时，若磁盘 d_0 的负载小于磁盘 d_1 的负载加上 *initialLoad*，则必定选择磁盘 d_0 上的副本。因此，随着 *initialLoad* 的不断增大，选择磁盘 d_1 上的副本的可能性不断下降，查询响应时间不断减少。但是，当 *initialLoad* 大到一定程度，查询响应时间会增加。其原因是即使磁盘 d_0 的负载已经比较重，还是会选择磁盘 d_0 上的副本。虽然可由磁盘 d_0 所在的节点执行读磁盘操作，但其所需时间已经大于由其他有 Impala 服务进程的节点执行对磁盘 d_1 远程读磁盘操作所需时间。

6.4.3 无并发查询请求情况下的适用于单表查询的调度方法对比实验

在 TPC-DS 20GB、40GB、60GB 和 80GB 数据集规模下，在无并发查询请求情况下，对比本文提出的适用于单表查询的调度方法和 Impala2.0 原有的调度方法对查询响应时间的影响，多次执行单表查询语句“select sum(ss_ext_sales_price) extended_price from store_sales”，记录查询响应时间并求平均值，集成了本文提出的适用于单表查询的调度方法的 Impala2.0 用集成后的 Impala 系统表示，集成前的 Impala 系统表示原有的 Impala2.0，实验结果如图 6.3 所示：

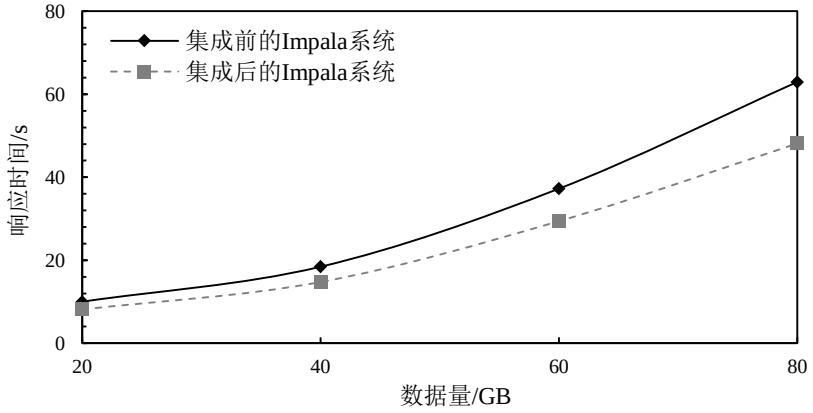

图 6.3 单表查询响应时间对比

由图 6.3 可知，集成后的 Impala 系统的查询响应时间的增长速度慢于集成前的 Impala 系统。当数据集规模达到 80GB 时，集成后的 Impala 系统的查询响应时间比集成前的 Impala 系统缩短了 23.5%。

在 TPC-DS 20GB、40GB、60GB 和 80GB 数据集规模下，在无并发查询请求情况下，对比本文提出的适用于单表查询的调度方法和 Impala2.0 原有的调度方法对读磁盘操作的执行时间的影响，多次执行单表查询语句“select sum(ss_ext_sales_price) extended_price from store_sales”，记录读磁盘操作的执行时间并求平均值，集成了本文提出的适用于单表查询的调度方法的 Impala2.0 用集成后的 Impala 系统表示，集成前的 Impala 系统表示原有的 Impala2.0，实验结果如图 6.4 所示：

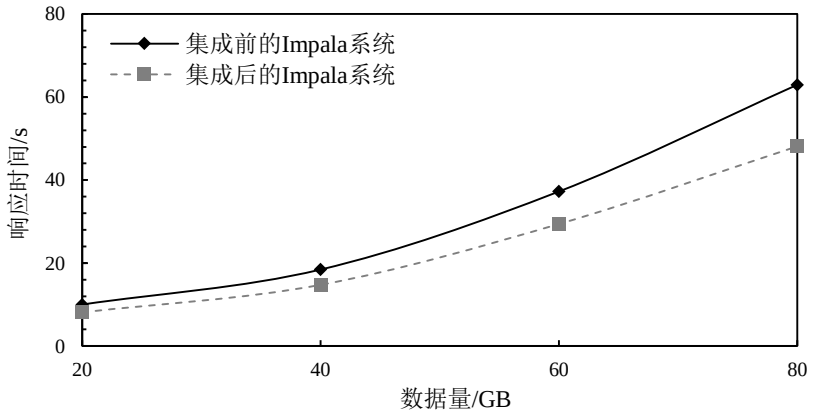

图 6.4 单表查询的读磁盘操作执行时间对比

由图 6.4 可知，集成后的 Impala 系统的读磁盘操作的执行时间的增长速度慢于集成前的 Impala 系统。当数据集规模达到 80GB 时，集成后的 Impala 系统的读磁盘操作的执行时间比集成前的 Impala 系统缩短了 32.19%。

因为最大流方法的目标就是最小化读磁盘操作的执行时间，而 Impala2.0 的调度方法是以任务量均衡为目标，所以集成后的 Impala 系统的读磁盘操作的执行时间的增长速度慢于集成前的 Impala 系统。因此，集成后的 Impala 系统的查询处理响应时间的增长速度慢于集成前的 Impala 系统。

在 TPC-DS 20GB、40GB、60GB 和 80GB 数据集规模下，在无并发查询请求情况下，对比本文提出的适用于单表查询的调度方法和 Impala2.0 原有的调度方法所需执行时间，多次执行单表查询语句“select sum(ss_ext_sales_price) extended_price from store_sales”，记录调度方法所需时间并求平均值，集成了本文提出的适用于单表查询的调度方法的 Impala2.0 用集成后的 Impala 系统表示，集成前的 Impala 系统表示原有的 Impala2.0，实验结果如表 6.4 所示：

表 6.4 单表查询调度方法所需时间对比（ms）

	20GB	40GB	60GB	80GB
集成前的 Impala 系统	1.53	2.69	3.86	4.95
集成后的 Impala 系统	20.35	119.59	238.17	343.63

由表 6.4 可知，集成后的 Impala 系统在单表查询处理过程中，调度方法本身

所需时间要高于集成前的 Impala 系统。而且随着数据量增加而明显升高。但单表查询调度所需时间占总查询时间的比例不到 1%，调度方法所需时间差异对总查询时间的影响可以忽略不计。集成后的 Impala 系统的单表查询调度方法所需时间高于集成前的 Impala 系统的原因如下，若用$|t|$表示查询涉及的数据块的数目，最大流方法的时间复杂度为 $O(\log|t|*|t|^3)$，而原有的 Impala 系统的调度方法的时间复杂度为 $O(|t|)$。

6.4.4 无并发查询请求情况下的适用于多表查询的调度方法对比实验

在 TPC-DS 20GB、40GB、60GB 和 80GB 数据集规模下，在无并发查询请求情况下，对比本文提出的适用于多表查询的调度方法和 Impala2.0 原有的调度方法对查询响应时间的影响，设置 *maxTime* 为 40，执行 q19、q42、q52 和 q55 查询语句，每个查询语句均执行多次得到平均响应时间，以 4 条查询语句的平均响应时间之和为总响应时间，集成了本文提出的适用于多表查询的调度方法的 Impala2.0 用集成后的 Impala 系统表示，集成前的 Impala 系统表示原有的 Impala2.0，实验结果如图 6.5 所示：

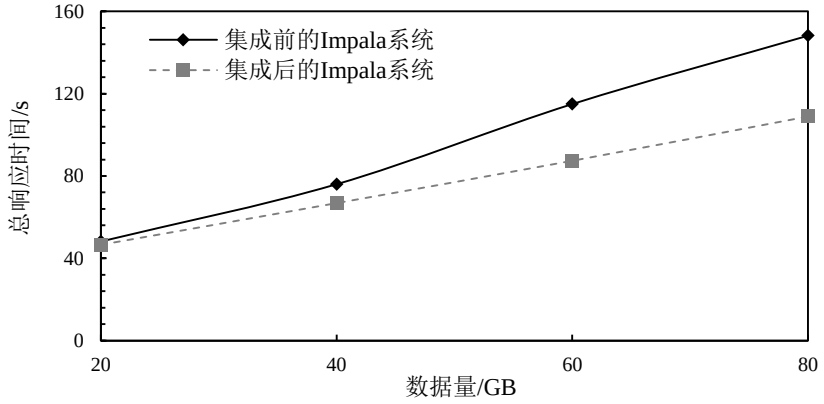

图 6.5 多表查询响应时间对比

由图 6.5 可知，集成后的 Impala 系统的查询响应时间的增长速度慢于集成前的 Impala 系统。当数据集规模达到 80GB 时，集成后的 Impala 系统的查询响应时间比集成前的 Impala 系统缩短了 26.46%。因为本文提出的适用于多表查询的调度方法的搜索空间包含 Impala2.0 原有的调度策略，又因为本文提出的适用于多

表查询的调度方法的搜索空间为 *maxTime*，Impala2.0 原有的调度方法的搜索空间为 1（以任务均衡为目标的贪心算法），所以，集成后的 Impala 系统的查询响应时间的增长速度慢于集成前的 Impala 系统。

在 TPC-DS 20GB、40GB、60GB 和 80GB 数据集规模下，在无并发查询请求情况下，对比本文提出的适用于多表查询的调度方法和 Impala2.0 原有的调度方法所需时间，设置 *maxTime* 为 40，执行 q19、q42、q52 和 q55 查询语句。每个查询语句均执行多次得到平均调度方法所需时间，以 4 条查询语句的平均调度方法时间之和为总调度方法所需时间，集成了本文提出的适用于多表查询的调度方法的 Impala2.0 用集成后的 Impala 系统表示，集成前的 Impala 系统表示原有的 Impala2.0，实验结果如表 6.5 所示：

表 6.5 多表查询调度方法所需时间对比（ms）

	20GB	40GB	60GB	80GB
集成前的 Impala 系统	1.52	2.39	3.64	4.28
集成后的 Impala 系统	166.66	303.04	430.00	562.20

由表 6.5 可知，集成后的 Impala 系统在多表查询过程中，调度方法本身所需时间明显高于集成前的 Impala 系统，但多表查询调度方法所需时间占总查询时间的比例不到 0.6%，调度方法所需时间差异对总查询时间的影响可以忽略不计。集成后的 Impala 系统调度方法本身所需时间明显高于集成前的 Impala 系统的原因是集成后的 Impala 系统的调度方法的搜索空间大于集成前的 Impala 系统。

6.4.5 无并发查询请求情况下的不同副本数的调度方法对比实验

在 TPC-DS 80GB 数据集规模下，在无并发查询请求情况下，从默认 3 个副本到每个 DataNode 均有副本，即副本数为 8，对比本文提出的查询调度方法和 Impala2.0 原有的调度方法对查询响应时间的影响，设置 *maxTime* 为 40，执行 q19、q42、q52 和 q55 查询语句，每个查询语句均执行多次得到平均响应时间，以 4 条查询语句的平均响应时间之和为总响应时间，集成了本文提出的适用于多表查询的调度方法的 Impala2.0 用集成后的 Impala 系统表示，集成前的 Impala 系统表示

原有的 Impala2.0，实验结果如图 6.6 所示：

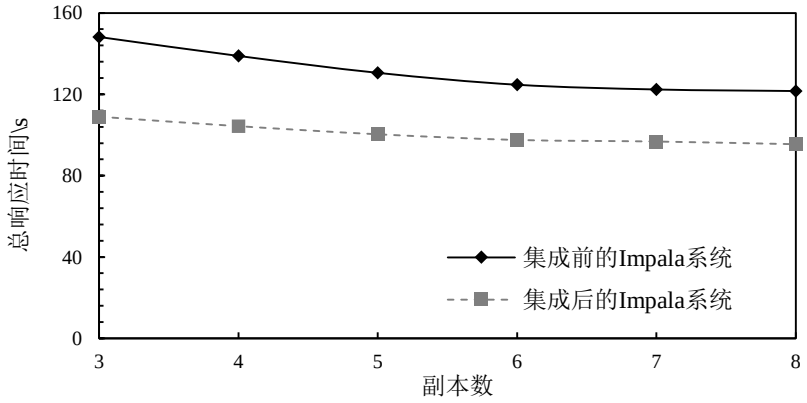

图 6.6 不同副本数下查询响应时间对比

由图 6.6 知，随着副本数的增多，集成前的 Impala 系统的查询响应时间和集成后的 Impala 系统的查询响应时间均在下降。集成前的 Impala 系统的查询响应时间随着副本数增多而缩短的原因是随着副本数增多，任务分配的均衡度进一步提高，而通信时间不一定增大，因此查询响应时间会减少。集成后的 Impala 系统的查询响应时间随着副本数增多的原因是随着副本数增多，候选调度方案增多，因此可能找到更优的调度方案。

6.4.6 有并发查询请求情况下的调度方法对比实验

在 TPC-DS 80GB 数据集规模下，在并发查询语句数目分别为 2、4、6 和 8 的情况下，对比本文提出的调度方法和 Impala2.0 原有的调度方法对查询响应时间的影响，设置 *maxTime* 为 40，执行 q19、q42、q52 和 q55 查询语句。每个查询语句均执行多次得到平均响应时间，以 4 条查询语句的平均响应时间之和为总响应时间，集成了本文提出的适用于多表查询的调度方法的 Impala2.0 用集成后的 Impala 系统表示，集成前的 Impala 系统表示原有的 Impala2.0，实验结果如图 6.7 所示：

由图 6.7 可知，当数据量为 80GB 时，随着并发查询语句数目的增加，集成后的 Impala 系统和集成前的 Impala 系统的查询响应时间均增长，但是集成后的 Impala 系统的查询响应时间的增长速度慢于集成前的 Impala 系统，当并发量为 8

时，集成后的 Impala 系统的查询响应时间比集成前的 Impala 系统缩短了 29.49%。。因为 Impala 服务进程采用多线程的方法支持并发，所以当并发的查询的数目增加时，集群的负载也增加。又因为本课题提出的调度方法考虑了集群的负载信息，所以集成后的 Impala 系统的查询响应时间的增长速度慢于集成前的 Impala 系统。

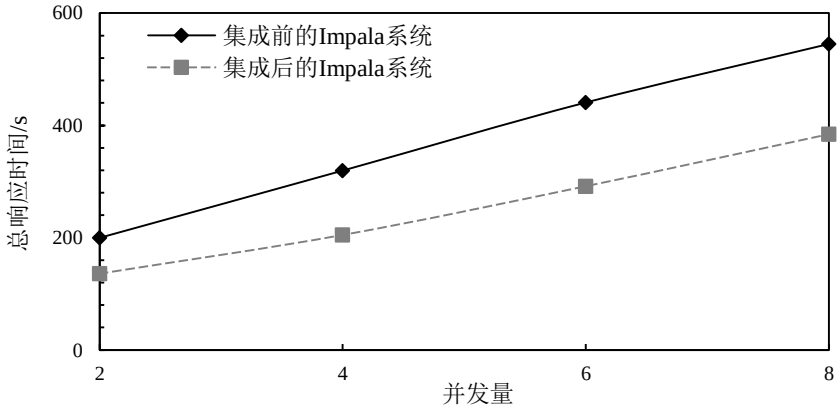

图 6.7 并发查询处理情况下查询响应时间对比

第7章 总结与展望

7.1 总结

大数据实时查询技术提供对大数据进行快速、交互式的查询。其为上层的OLAP（Online analytical processing）和商业智能提供支持。若大数据实时查询的延迟较长，则必会影响上层应用的用户体验。因此，缩短大数据实时查询的平均响应时间是大数据实时查询技术的一个重要的方面。

本课题深入分析了现有的大数据实时查询系统 Impala2.0，针对 Impala2.0 存在的在并行调度的过程中仅以任务量均衡为目标，不一定能保证查询响应时间较短的问题，在总结代价模型和并行调度方法两方面的研究工作的基础上，提出了基于副本选择的并行调度方法及 Impala 查询处理代价模型。本文的主要工作包括：

1) 研究了查询处理并行调度任务模型、代价模型和并行调度方法三方面的相关工作，之前的研究者已对并行调度的任务模型进行了归纳和分类。本文主要对代价模型和并行调度方法进行归纳和分类。本文将代价模型分为两类，一类是基于代价函数的代价模型，另一类是基于机器学习的代价模型。本文将并行调度方法分为非自适应的调度方法和自适应的调度方法两类。其中非自适应的调度方法又进一步分为操作树划分、粗粒度并行和启发式 3 类。但是，本课题调研的已有的调度方法均未考虑数据文件分块，同时每个块有多副本的情况；
2) 深入分析了 Impala 实时查询系统的体系架构和执行流程，特别分析了 Impala2.0 的并行调度模块的流程和 Impala2.0 的统计信息及选择度计算模块；
3) 提出基于副本选择的大数据实时查询处理并行调度方法，考虑数据文件分块，同时每个块有多副本的情况，在调度过程中综合考虑查询计划、数据在集群的分布情况、数据本地性和节点磁盘负载情况；

4) 构建 Impala 并行处理代价模型，综合考虑任务量、并行执行、CPU 性能、磁盘性能和磁盘负载，估计中间结果以提高代价模型的准确度；
5) 将调度方法集成到 Impala2.0 系统，对提出的基于副本选择的大数据实时查询处理并行调度方法的有效性进行实验验证，对方法的复杂度进行分析，对实验结果的原因进行分析。

综上所述，本课题的贡献在于提出 Impala 查询处理代价模型和基于副本选择的调度方法，将代价模型和调度方法在 Impala2.0 上集成与实现，通过实验结果验证了方法的有效性。缩短了大数据实时查询的平均响应时间，有助于改善上层应用的用户体验。

7.2 展望

本课题提出的基于副本选择的调度方法还存在很大的拓展空间。在接下来的研究工作中，可以围绕以下几个方向进行展开：

第一，研究更高效的并行调度方法。本课题提出的并行调度方法根据查询涉及的表的数目将所有查询分为单表查询和多表查询。针对多表查询，本课题结合 Impala 并行处理代价模型搜索近似最优解。本课题将搜索的次数即 *maxTime* 作为参数，后续研究可考虑根据查询涉及的表的大小动态调整 *maxTime*。

第二，研究更准确的并行处理代价模型。本课题提出的 Impala 并行处理代价模型中未对聚集操作构建模型，后续研究可考虑为 Impala 的聚集操作构建模型。

第三，研究更准确的中间结果估计方法。本课题为数据库表构建直方图以提供较准确的中间结果估计。因为查询处理过程中的操作是在集群中若干个节点上的并行执行的，参与某个节点上的某个操作的数据只是数据库表的一部分，所以相同的谓词对节点上的部分数据库表的选择度可能不同于对完整的数据库表的选择度。后续研究可针对节点上的部分数据库表构建直方图。

参考文献

[1] A. Hameurlain, F. Morvan. Evolution of Query Optimization Methods[J]. LNCS, 2009, 5740, 211-242.

[2] W. Hasan, R. Motwani. Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism[C]. VLDB, 1994: 36-47.

[3] T. Ibaraki, T. Kameda. On The Optimal Nesting Order for Computing N-relational Joins[J]. TODS, 1984, 9(3), 482-502.

[4] C. Chekuri, W. Hasan, R.Motwani. Scheduling Problems in Parallel Query Optimization[C]. PODS, 1995: 255-265.

[5] M. N. Garofalakis, Y. E. Ioannidis. Multi-dimensional Resource Scheduling for Parallel Queries[C]. SIGMOD, 1996: 365-376.

[6] G. M. Lohman, C. Mohan, L. M. Haas, et al. Query processing in R*[M]. Springer Berlin Heidelberg, 1985.

[7] M. Ozsu, P. Valduriez. Principles of database systems(3rd)[M]. Springer-Verlag New York Inc, 2011.

[8] A. N. Wilschut, J. Flokstra, P. M. G.Apers, et al. Parallel Evaluation of Multi-join Queries[C]. SIGMOD, 1995: 115-126.

[9] S. Melnik, A. Gubarev, J. J. Long, et al. Dremel: Interactive Analysis of Web-Scale Datasets[C]. VLDB, 2010: 330-339.

[10] Cloudera Installing and Using Impala [EB/OL]. http://www.cloudera.com/content/support/en/documentation/cloudera-Impala/cloudera-Impala-documentation-v1-latest.html.

[11] Apache Spark[EB/OL]. https://spark.apache.org/.

[12] Apache Drill[EB/OL]. http://incubator.apache.org/drill/.

[13] K. Shvachko, H. Kuang, S. Radia, et al. The Hadoop Distributed File System[C] MSST, 2010: 1-10.

[14] J. Dean, S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters[J]. CACM, 2008, 51(1): 107-113.

[15] A. Thusoo, J. S. Sarma, N. Jain, et al. Hive: A Warehousing Solution Over a Map-Reduce Framework[J]. PVLDB, 2009, 2(2): 1626-1629.

[16] L. Chang, Z. Wang, T. Ma, et al. HAWQ: A Massively Parallel Processing SQL Engine in Hadoop[C]. SIGMOD, 2014: 1223-1234.

[17] Pivotal Greenplum Database [EB/OL]. http://www.pivotal.io/big-data/pivotal-greenplum-database.

[18] M. A. Soliman, L. Antova, V. Raghavan, et al. Orca: a Modular Query Optimizer Architecture for Big Data[C]. SIGMOD, 2014: 337-348.

[19] S. Cluet, G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products[C]. ICDT, 1995: 54-67.

[20] T. Nykiel, M. Potamias, C. Mishra, et al. MRShare: sharing across multiple queries in MapReduce[J]. PVLDB, 2010, 3(1-2): 494-505.

[21] A. Okcan, M. Riedewald. Processing theta-joins using MapReduce[C]. SIGMOD, 2011: 949-960.

[22] H. Herodotou, S. Babu. Profiling, what-if analysis, and cost-based optimization of MapReduce programs[J]. PVLDB, 2011, 4(11): 1111-1122.

[23] A. Gruenheid, E. Omiecinski, L. Mark. Query optimization using column statistics in hive[C]. IDEAS, 2011: 97-105.

[24] D. Kossmann. The State of the Art in Distributed Query Processing[J]. (CSUR), 2000, 32(4): 422-469.

[25] S. Khoshafian, P. Valduriez. Sharing persistence and object-orientation: A database perspective[C]. DBPL, 1987: 181-205.

[26] S. Ganguly, A. Goel, and A. Silberschatz. Efficient and Accurate Cost Models for Parallel Query Optimization[C]. PODS, 1996: 172-181.

[27] S. Shankar, R. Nehme, J. Aguilar-Saborit, et al. Query Optimization in Microsoft SQL Server PDW[C]. SIGMOD, 2012: 767-775.

[28] S. F. M. Sampaio, N. W. Paton, J. Smith, and P. Walson. Measuring and Modelling the Performance of a Parallel ODMG Compliant Object Database

Server[J]. Concurrency Computat. Pract. Exper., 2006, 18(1): 63-109.

[29] R. J. Lipton, J. F. Naughton, D. A. Schneider. Practical selectivity estimation through adaptive sampling[R]. 1990.

[30] P. J. Haas, A. N. Swami. Sampling-based selectivity estimation for joins using augmented frequent value statistics[C]. ICDE, 1995: 522-531.

[31] P. J. Haas, J. F. Naughton, S. Seshadri, et al. Sampling-based estimation of the number of distinct values of an attribute[C]. VLDB. 1995: 311-322.

[32] W. Sun, Y. Ling, N. Rishe, et al. An instant and accurate size estimation method for joins and selections in a retrieval-intensive environment[C]. SIGMOD Record. 1993, 22(2): 79-88.

[33] C. M. Chen, N. Roussopoulos. Adaptive selectivity estimation using query feedback[C]. SIGMOD, 1994: 161-172.

[34] Y. E. Ioannidis, S. Christodoulakis. Optimal histograms for limiting worst-case error propagation in the size of join results[J]. TODS, 1993, 18(4): 709-748.

[35] Y. E. Ioannidis, V. Poosala. Balancing histogram optimality and practicality for query result size estimation[J]. ACM SIGMOD Record, 1995, 24(2): 233-244.

[36] V. Poosala V, P. J. Haas, Ioannidis Y E, et al. Improved histograms for selectivity estimation of range predicates[J]. ACM SIGMOD Record, 1996, 25(2): 294-305.

[37] A. Ganapathi, H. A. Kuno, U. Dayal, et al. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning[C]. ICDE, 2009: 592-603.

[38] M. Akdere, U. Cetintemel, M. Riondato, et al. Query Performance Modeling and Prediction[C]. ICDE, 2012: 390-401.

[39] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal. Performance Prediction for Concurrent Database Workloads[C]. SIGMOD, 2011: 337-348.

[40] A. Temehchi, M. Ghodsi. Pipelined Operator Tree Scheduling in Heterogeneous Environments[J]. JPDC, 2003 63(6): 630-637.

[41] C. Odubasteanu, C. Munteanu. Parallel Query Optimization: Pipelined Parallelism Scheduling And Golden Number[J]. U.P.B. Sci. Bull., Series C, 2009, 71(3): 105-120.

[42] H. I. Hsiao, M. S. Chen, P. S. Yu. On Parallel Execution of Multiple Pipelined Hash Joins[C]. SIGMOD, 1994: 185-196.

[43] S. Ganguly, W. Wang. Optimizing Queries for Coarse Grain Parallelism[R]. New Grunswick, Univ. of Rutgers, 1993.

[44] E. Rahm, R. Marek. Dynamic Multi-Resource Load Balancing in Parallel Database Systems[C]. VLDB, 1995: 395–406.

[45] B. Nam, M. Shin, H. Andrade, et al. Multiple Query Scheduling for Distributed Semantic Caches[J]. JPDC, 2010, 70(5): 598-611.

[46] A. A. Kolalei, and M. Ahmadzadeh. The Optimization of Running Queries in Relational Databases Using ANT-Colony Algorithm[J]. IJDMS, 2013, 5(5): 1-8.

[47] W. Hong. Exploiting inter-operation parallelism in XPRS [C]. SIGMOD, 1992: 19-28.

[48] M. Stonebraker, R. H. Katz, D. A. Patterson, et al. The Design of XPRS[C]. VLDB, 1988: 318-330.

[49] N. kabra, D. J. Dewitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans[C]. SIGMOD, 1998: 106-117.

[50] D. J. DeWitt, N. Kabra, J. Luo, et al. Client-server paradise[C] VLDB, 1994: 558-569.

[51] E. Rahm, R. Marek. Analysis of Dynamic Load Balancing Strategies for Parallel Shared Nothing Database Systems[C]. VLDB, 1993: 182-193.

[52] M. Mehta, D. J. DeWitt. Managing Intra-operator Parallelism in Parallel Database Systems[C]. VLDB, 1995: 529-548.

[53] C. B. Walton, A. G. Dale, R. M. Jenevein. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins[C]. VLDB, 1991: 537-548.

[54] D. J. DeWitt, J. F. Naughton, D. A. Schneider, et al. Practical skew handling in parallel joins[C]. VLDB, 1992: 27-40.

[55] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, et al. Flux: An Adaptive Partitioning Operator for Continuous Query Systems[C]. ICDE, 2003:25–36.

[56] L. T. Chen, D. Rotem. Optimal Response Time Retrieval of Replicated Data[C]. SIGACT-SIGMOD-SIGART, 1994: 36–44.

[57] A. S. Tosun Multi-site Retrieval of Declustered Data[C]. ICDCS, 2008: 486–493.

[58] N. Altiparmak, A. S. Tosun. Integrated Maximum Flow Algorithm for Optimal Response Time Retrieval of Replicated Data[C]. ICPP, 2012: 11-20.

[59] G. M. Amdahl Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities[C]. AFIPS, 1967: 483-485.

[60] TPC-DS[EB/OL]. http://www.tpc.org/tpcds.

[61] M. Kornacker, A. Behm, V. Bittorf, et al. Impala: A Modern, Open-Source SQL Engine for Hadoop[C]. CIDR, 2015: 1-10.

攻读硕士学位期间主要的研究成果

[1] 赵宇亮, 陈岭, 杨谊, 马骄阳, 吴勇, 王敬昌. 基于副本选择的 Impala 并行查询执行调度[J].计算机研究与发展(增刊，第 31 届中国数据库学术会议 NDBC), 2014 : 118-125.

[2] 浙江大学, 一种用于分布式文件系统中大数据查询的调度方法: 中国, 申请号 2014106706969 申请日期: 2014.11.20.

致谢

时光如白驹过隙，让我来不及准备就悄然流逝。不知不觉，已在美丽的浙大玉泉校区度过了两年半的美好时光。回首研究生科研经历，我要对老师、同学、和所有帮助过的人表示最衷心的谢意。

首先，我要由衷的感谢我的导师陈岭老师，本课题是在陈老师的耐心指导下完成的。在研究过程中，陈老师从课题选择到文献查找，再到研究方案、实验方案直至论文的审核，都给了我很多宝贵的意见和帮助。陈老师严谨求是的治学态度、渊博的知识面和清晰的科研思路给我留下了深刻的印象。在研究学习期间，我在陈老师的直接指导下对大数据实时查询领域及并行调度的一些问题进行了研究和探索。陈老师给我的学术研究工作带来了莫大的帮助和支持。

此外，我要感谢实验室中的全体师兄师姐师弟师妹们（涂鼎，吕明琪， Ibrar Hussain，王礼文，刘荣游，涂游，唐燕琳，蔡晓东，何旭峰，孔星，周强，侯仓健，邵维，范阿琳，范长军，郭浩东，杨谊，蔡雅雅，马骄阳，徐振兴，袁翠丽，余小康，沈延斌，王俊凯，顾伟东，韩保礼，林言，吴晓杰，应鸳凯，余宙，王耀光，孔德江，郑福真，申亚鹏，刘坚，金鑫，白建飞，李旭，蒋静远等）对我研究工作的帮助和支持。和他们探讨问题让我成长的更快。

另外，我要感谢鸿程研发中心的同事和朋友，感谢他们帮助我提高工程实践能力，感谢他们帮助我了解企业实际使用的大数据分析平台。

同时，我要感谢我的父母，感谢他们对我的养育之恩，感谢他们对我生活的关心和对我学习的支持。他们的鼓励是我最大的精神动力。

最后，我要感谢各位评审老师审阅本文并提出合理而宝贵的意见。

赵宇亮

于浙江大学计算机科学与技术学院

2015-01-23