浙江大学

本科生毕业设计
开题报告

学生姓名：汤利康

学生学号：3110102975

指导教师：陈岭

年级与专业：11级计算机科学与技术

所在学院：计算机学院

一、题目： 基于直方图方法估计中间结果的 Impala 系统优化

二、指导教师对开题报告、外文翻译和中期报告的具体要求：

1 开题报告要求从现有工作的不足或具体应用需求中导出毕业设计的立题依据，确定具体的目标，提炼可能的研究、开发内容，大致确定毕业设计各组成部分、相应关键技术及技术路线，制定毕业设计时间安排。要求开题报告字数在3500字以上。

2 外文翻译要求选择和毕业论文题目相近，且较经典的外文文献。翻译过程中对出现的专业词汇要逐一掌握，理解文中介绍系统、技术和方法等的具体原理。要求翻译后的文档语言简练、通顺，描述准确。要求外文翻译字数在3000字以上。

3 中期实习报告要求全面梳理前期实习工作，总结已完成的研发工作及过程中出现的问题，明确下一阶段的工作目标、内容及需重点解决的关键问题，制定相应技术路线，并根据后期实习工作内容的变化对实习工作计划进行调整。

指导教师（签名）

年 月 日

毕业设计开题报告、外文翻译和中期报告考核

导师对开题报告、外文翻译和中期报告评语及成绩评定：

成绩比例	开题报告 占（20%）	外文翻译 占（10%）	中期报告 占（10%）
分 值			

导师签名

年　　月　　日

答辩小组对开题报告、外文翻译和中期报告评语及成绩评定：

成绩比例	开题报告 占（20%）	外文翻译 占（10%）	中期报告 占（10%）
分 值			

开题报告答辩小组负责人（签名）

年　　月　　日

-

目录

本科毕业设计开题报告

1. 项目背景

1.1 Impala 大数据实时查询系统

Impala 是 Cloudera 公司主导开发的新型大数据查询系统，它提供 SQL 语义，支持对存储在 HDFS 和 HBase 的数据进行快速地、实时地查询。Impala 使用与 Hive 相同的数据存储平台，元数据，SQL 语法（Hive SQL），ODBC 驱动程序和用户界面（Hue Beeswax）。Impala 还提供了一个面向批量的实时查询平台。

在 Cloudera 的测试中，Impala 的查询效率相比于 Hive 有数量级的提升。从技术角度分析，Impala 系统性能提升如此之大主要有以下几点原因：

1) Impala 不会把查询中间结果写入磁盘而避免了大量的 I/O 操作。
2) Impala 系统通过自带的服务进程来进行作业调度，省掉了 MapReduce 作业启动的开销。
3) Impala 抛弃 MapReduce 这个不太适合 SQL 查询的范式，而是使用 MPP 并行数据库思想，以便做更多的查询优化。
4) 使用了 Data Locality 的 I/O 调度机制，避免了网络数据传输带来的成本。

Impala 开源项目自发布以来，以其优秀的性能，受到了学术界和工业界的广泛关注。目前，Cloudera 已经发布了 Impala2.0。本文中所介绍的 Impala 特点和将来所做的系统优化都针对于 2.0。

1.2 系统架构

Impala 是基于 Hadoop 的分布式的、高并发的 SQL 引擎。其主要架构如图 1.1 所示。

从图中可以得出，一个 Impala 系统主要由三个服务进程所构成。

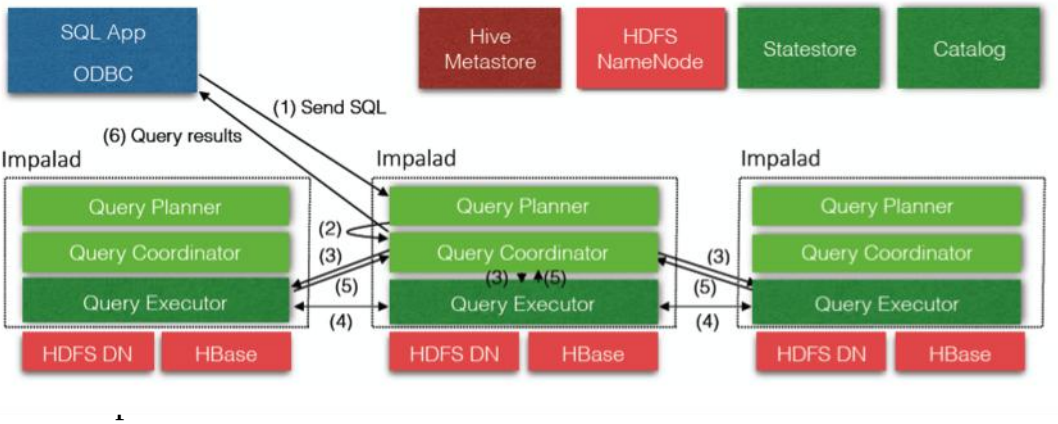


图 1.1 Impala 系统架构图

1) Impalad 服务进程

Impalad 服务负责从 Client 接收查询请求，生成查询计划分发给其它 Impalad，最后接收整合其它 Impalad 进程返回的结果，将结果返回给 Client。它主要由查询计划器、查询协调器和查询执行器组成。其中，查询计划器负责将接收到的 Client 端查询请求转换为查询计划。这其中包括了查询计划树的生成，查询优化，查询分片等步骤。查询协调器根据查询计划以及集群状态分配查询任务，并最终收集各个节点返回的查询结果返回给 Client。而查询执行器负责查询计划的执行。

2) Statestore 进程

Statestore 进程负责监控整个 Impala 系统。每个 Impalad 服务进程在启动时都会向 Statestore 注册，而当集群中的某一个节点发生故障时，Statestore 将该故障信息通知给集群中所有的节点。这样，系统中的查询任务就不再会被分配给故障节点，提高了系统的鲁棒性。

3) Catalog 服务

Catalog 服务负责管理系统的元数据信息。Impala 会缓存元数据信息。当某个 Impalad 服务进程处理 DDL 请求时，其它 Impalad 进程的元数据信息已经过时，这时就需要 Catalog 进程更新其它 Impalad 进程的元数据信息。

1.3 Catalog 服务原理

本文 1.2 节中已经提到 Catalog 的主要功能。因为接下来的优化主

要是针对 Impala 系统中的 Catalog 模块。所以在以下篇幅中将介绍 Catalog 的原理以及目前存在的一些不足。

Catalog 主要分为前端和后端两部分。

在每一个 Impalad 服务进程的前端部分，维护了如图 1.2 所示结构。

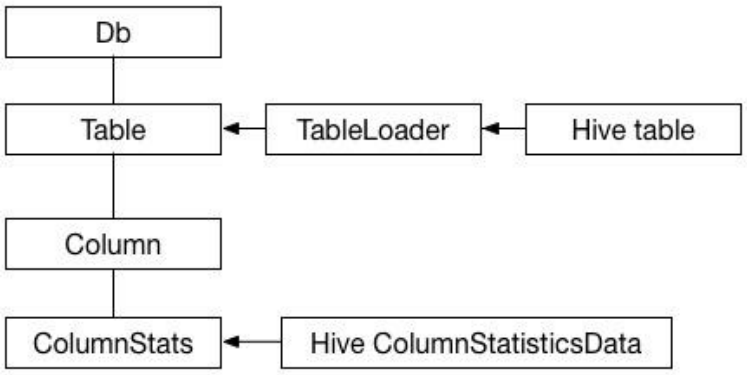


图 1.2　Catalog 服务元数据结构图

当某一个 Impalad 服务初始化时，会初始化一个 Catalog 对象，在其对象中包含了一个数据库缓存，接着载入 Hive 或者 HBase 中的数据库元数据信息。在初始化时，系统会为数据库中每张表创建一个入口，但是并不载入具体信息。当有某一个 DDL 请求涉及到相关 Table 时，Impalad 会使用 TableLoader 类载入 Hive 中表的具体元数据信息。表的具体元数据信息包括每一列的信息，包括列名，列类型，列位置等。列中的 ColumnStats 对象则是从 Hive 的 ColumnStatisticsData 中获得的对列数据层次的统计信息，包括列的平均容量，互不相同的值的个数等。

而 Catalog 的后端部分只存在于一个节点上。它的主要功能是负责搜集各台主机上的 Catalog 更新信息。它主要维护了一个 Catalog 的版本号，任何节点上的 Catalog 改变都会引起其版本的改变。另外在后端 Catalog 上运行了一个 GatherCatalogUpdatesThread 线程来监控 Catalog 版本号，当发现后端版本号发生变化时，该线程会获取发生变化的元数据对象列表，将变化部分广播给整个系统上的每个节点。

1.4 代价模型与中间结果估计

在制定查询计划时，查询优化是必不可少的，而查询优化的关键点之一是建立有效的代价模型。其中基于代价函数的代价模型用代价函数来表示查询处理的代价。

而为了提高代价模型的准确度，需要估计查询处理的中间结果的大小。直方图方法能够提供较精准的中间结果估计。与其它方法，如抽样方法和参数方法相比，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。

对于某一列 X 的直方图，它将列值的数据分布范围划分为β个子集。每一个子集被称为一个桶(bucket)。桶高度为这个子集的数据频数。而分桶规则由具体的直方图方法决定。

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

表 1.3 直方图

如表 1.3 的直方图表示某一列，该列从 102 到 106 有 50 个值，从 109 到 112 有 70 个值，从 120 到 123 有 40 个值。

而在直方图方法中，Maxdiff(V,A)方法误差率较小。Maxdiff(V,A)在较大差值的值中间插入β-1 个桶边界以分为β个桶，因此它会将较小差值的值分入一个桶中。而在具体的创建中，我们只需要寻找两个差值较小的相邻桶合并即可。这里的差值主要比较桶宽(highBound-lowBound)和平均频数(Cardinality/(highBound-lowBound))，桶宽的优先级较高，在桶宽相同的情况下再考虑平均频数的差值。换言之，在桶数达到β个时，合并桶宽和平均基数对差距最小的两个相邻的桶。

2. 目标和任务

在现有 Impala 系统 Catalog 服务的基础上，增加直方图信息统计模块，以提高代价模型的准确度，为查询优化提供更大的空间。

直方图信息统计模块针对数据表中的列建立直方图，提供给查询优化器使用。为达到以上目标。我们将需要完成的任务分为三个阶段。

1) 熟悉 Impala 系统现有的 Catalog 服务的基础架构和工作流程。
2) 实现 Maxdiff（A,V）直方图创建算法。使用 JDBC 连接数据库获得

列数据，并创建直方图。

3) 将第二阶段中创建的直方图创建模块集成到 Impala 系统中，使 Impala 系统能够创建和维护直方图信息，提供给查询优化器使用。

3. 可行性分析

3.1 课题开展的意义

通过以上的描述，可以得知目前 Catalog 只维护了 Db、Table 和 Cloumn 信息。但是对于一些查询优化方法来说，仅仅有这些信息还是不够的。如果能够在已有的 Catalog 的基础上增加一些统计信息，无疑会对查询优化提供更大的支持。而基于列做直方图信息统计用于估计中间结果，能够在很大程度上提高代价模型的准确性。因此，本课题实现直方图方法查询中间结果估计，并将其集成到 Impala 大数据实时查询系统中去对 Impala 查询优化效率的提升是十分有意义的。

3.2 技术可行性

3.2.1 直方图的创建

本课题通过 JDBC 连接数据库获取某一列的数据，并通过 Maxdiff(V,A)直方图算法创建直方图。因此在创建直方图上是没有问题的。

3.2.2 直方图数据的维护和共享

在 Impala 系统的一般存在很多的节点。在每一个数据节点上都可能存在着一部分数据。为了提高能够使直方图创建的效率，根据 Data-Locality 原则，我们将直方图的创建放在该列所在的数据节点进行以减少网络传输。因此在每一个数据节点上都可能存在着一部分直方图信息。这些直方图信息通过 Catalogd 服务进程统一维护和请求。通过 Thrift 框架进行通信。因此直方图数据在整个 Impala 系统中共享和维护是可行的。关于 Thrift 框架，它用来进行可扩展且跨语言的服务的开发。它结合了功能强大的软件堆栈和代码生成引擎，以构建在

C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, and OCaml 这些编程语言间无缝结合的、高效的服务。在 Impala 工程中就使用了 Thrift 实现了后端和前端之间的通信。

3.2.3 对原有系统的影响

Impala 系统是一个开源的大数据实时查询引擎，使用 Github 维护源代码。因此我们将直方图统计模块进程到 Impala 系统中是完全可行的。

Impala 系统注重实时性。为此，需要注意建立直方图对查询造成影响。因为只有写操作才会影响列数据，所以只在写操作时才更新相关列的直方图。另外，直方图的创建也是在写操作完成之后再后台进行。所以直方图的创建对原有系统的实时性影响可以忽略不计。

4. 初步技术方案和关键技术考虑

4.1 直方图模块整体架构

本课题将直方图统计模块集成到 Impala 系统的 Catalog 中，因此其架构也基于 Catalog 服务架构。基本架构如下图所示。

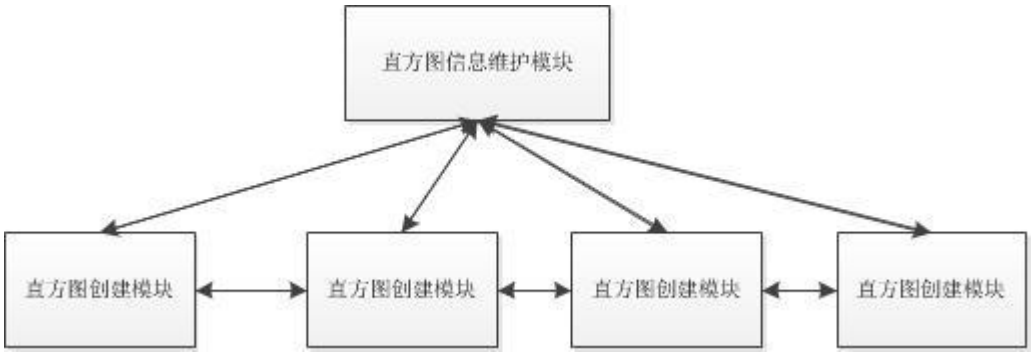


图 4.1 直方图统计模块架构图

类似于 Impala 系统中 Statestore 服务和 Impalad 进程的关系，直方图信息维护模快只存在于一个节点上，负责维护整个系统中直方图的信息，它维护了一张表，记录了哪些列的直方图已经被创建，被那个节点创建等信息。而每一个数据节点都会有一个直方图创建模块，负

责创建本数据节点所拥有列数据的直方图的创建和维护。之所以这样做，是出于减少数据网络传输的考虑。

直方图维护模块和创建模块之间传递直方图是否创建，创建者为哪个节点，创建某一列的直方图等信号。而直方图创建模块之间则传递具体的直方图数据。

图中直方图信息维护模块主要提供以下几个接口：

1) IsHistogramBuilt:某一列的直方图是否已经创建。
2) BuildHistogram:创建某一列的直方图。该接口并不直接创建直方图，而是调用相关节点直方图创建模块的 BuildHistogram 接口。需要参数，表名，列名，及直方图创建完成之后的回调。直方图请求者通过回调可以从直方图创建者处获得直方图信息。该方法需要通过 Hive 获得请求创建直方图的列数据所在节点，并调用该节点直方图创建模块的接口创建直方图。
3) GetHistogramBuilder:获得直方图创建节点信息。

直方图创建模块负责创建直方图，其主要提供以下接口：

1) Buildhistogram:该接口被直方图维护模块的 BuildHistogram 接口调用，负责创建某列的直方图。通过 JDBC 访问数据库获得数据，根据数据创建直方图。创建完成后通过回调通知维护统计模块其已经完成该列直方图的创建。
2) GetHistogram:返回直方图数据。

4.2 直方图统计流程

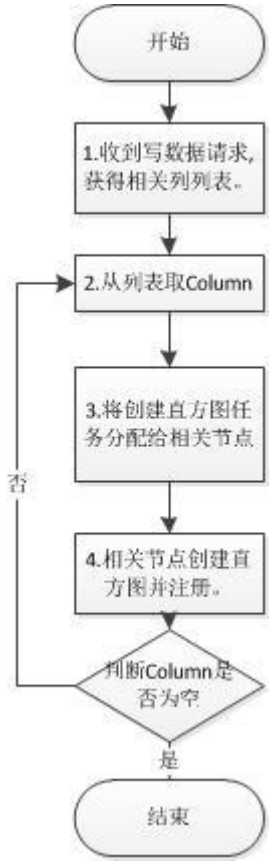


图 4.2 直方图统计模块流程图

相关步骤说明：

1) 在步骤一中，只有在写操作时数据直方图才可能会发生变化，所以只在写数据请求时才触发创建直方图。解析器将写操作解析获得相关的 Column 列表。
2) 在步骤二中，开始遍历上面获得的列表，为每一列建立直方图创建任务。
3) 在步骤三中，根据 Data-locality 原则，将直方图的创建任务发送给该列所在数据节点直方图创建模块执行。
4) 在步骤四中，当某一个直方图建立完成后，会向通过回调向维护模

块报告已经建立某列的直方图信息。而维护模块维护这些信息，当某列的直方图需要被使用时，即可向维护模块询问在哪个节点拥有该列直方图信息。

上述流程除了步骤四在直方图创建模块执行外，其它步骤都在直方图信息维护模块执行。

4.3 直方图创建方法

4.3.1 直方图创建流程图

本课题主要使用 Maxdiff(V, A)直方图。创建直方图的流程图如下所示。该流程在直方图创建模块执行，是 4.2 中流程步骤四的具体实现。

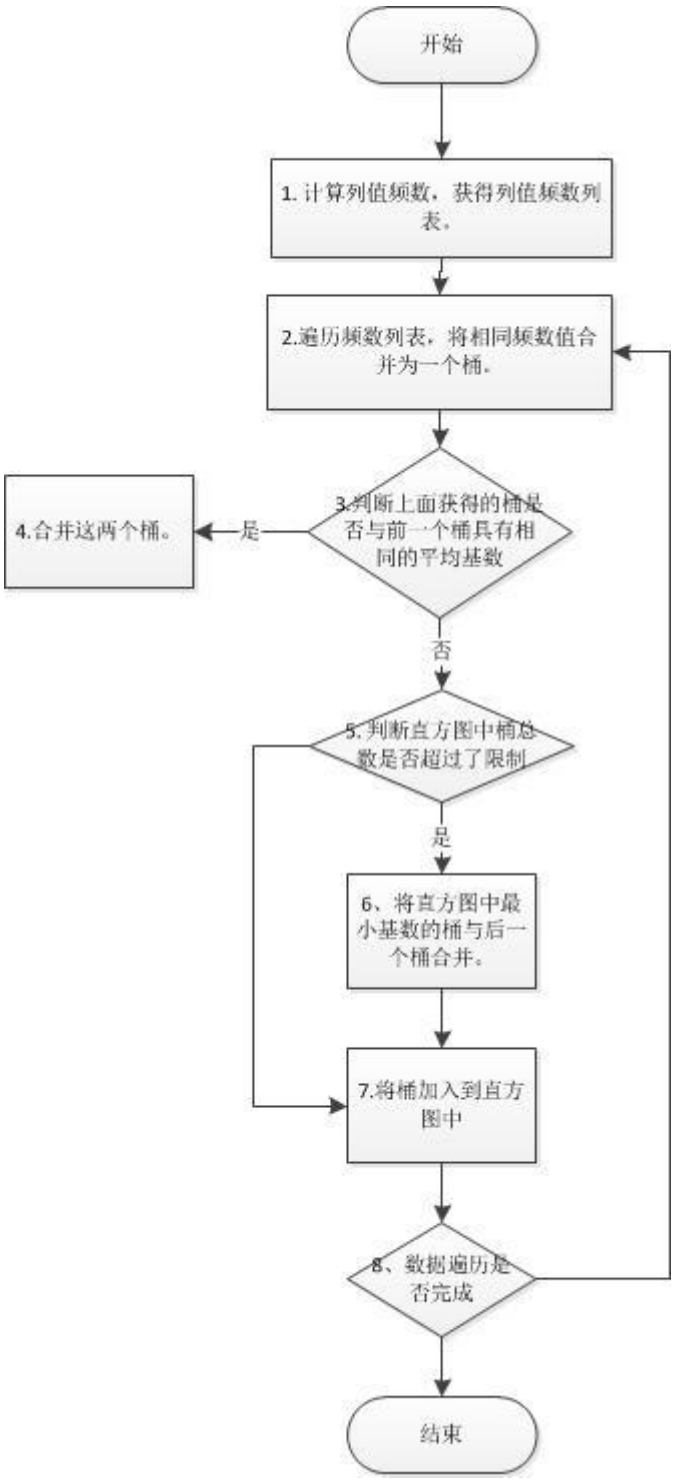


图 4.3 直方图创建算法流程图

4.3.2 算法说明

以下将配合上图做各步做详细说明。

1) 在步骤一中，根据列数据获得列值频数列表。即对列数据中的值进行计数。
2) 在步骤二中，遍历列频数列表，将相同频数的列值合并为一个桶。当发现不同频数的列值时，进入第三步。
3) 步骤三中，判断当前桶的平均基数是否与上一个相邻桶相同，如果是进入第四步，否则进入第五步。这里的平均基数为桶总频数和除以桶宽。
4) 步骤四中，将当前桶与上一个相邻桶合并。进入第七步。
5) 步骤五中，判断当前直方图中的桶数是否达到最大桶数限制，如果是进入第六步将直方图中最小桶和其后桶合并。否则直接进入第七步将桶加入直方图。这里的桶数最大值为 Maxdiff 直方图定义中的 β。
6) 步骤八中，将当前桶设为上一个桶，判断列表遍历是否完成，如果没有完成返回第二步，否则结束。

5. 预期工作结果

1) 实现 Maxdiff（A,V）直方图算法。能够根据数据库中获取到的列数据创建直方图。
2) 将直方图的创建和维护集成到 Impala 系统的 Catalog 中。能够较为快速地创建和维护直方图信息，提供给查询优化器使用。

6. 进度计划

时间范围	完成工作	备注
4 月 10 日前	熟悉相关代码，完成总体设计。	
4 月 20 日前	完成直方图创建算法，能够通过 JDBC 查询某一列数据，创建直方图。	
5 月 10 日前	将直方图创建集成到 Impala 的 Catalog 中。能够根据查询请求创建和更新直方图。	直方图的创建由直方图列所在数据节点执行。
5 月 20 日前	完成毕业设计报告	
5 月 30 日前	准备答辩	

本科毕业设计外文翻译

Impala：一个 Hadoop 的现代开源 Sql 引擎

1. 摘要：

Cloudera 公司的 Impala 是一个为 Hadoop 数据处理环境所构建的现代开源 MPP SQL 引擎。不同于 Apache 的 Hive 批处理架构，Impala 为在 Hadoop 上的以 BI/分析读操作为主要操作的查询提供了低延迟和高并发。这篇 Paper 从用户的观点来陈述 Impala，并提供了对 Impala 构成，主要组成部件的概览。通过与在 Hadoop 上的主流 S QL 系统的比较，这篇 Paper 展示了 Impala 的优秀性能。

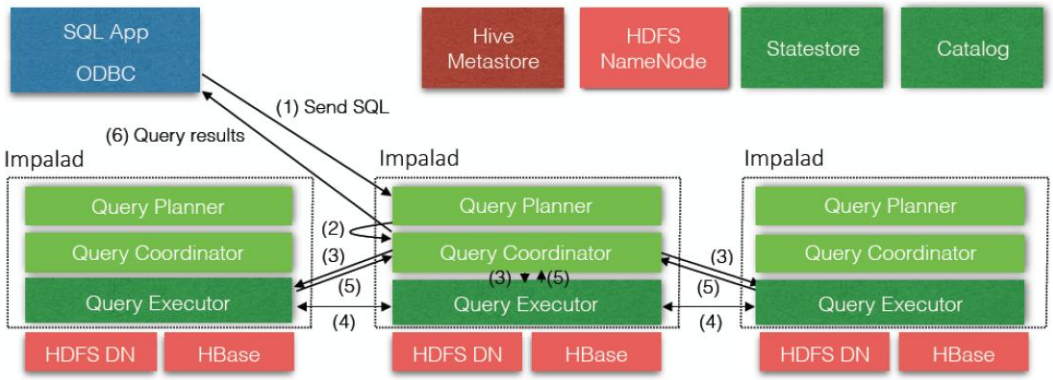
2. 介绍

Impala 是一个开源的，充分整合，代表了先进水平的 Mpp Sql 查询引擎。它专门被设计来平衡 Hadoop 的灵活性和可扩展性。Impala 的设计目标有三，首先为了实现熟悉的 SQL 支持。其次为了提供在传统分析型数据库上的多用户性能的基础上实现 Apache Hadoop 的可扩展性和灵活性。第三是为了支持 Cloudera Enterprise 上的工业级别的安全性和管理延生。

3. 结构

Impala 是一个运行在 Hadoop 簇上成千上百台主机上的高度并行的查询引擎。不同于传统的关系型数据管理系统那样查询处理和存储引擎都是一

个紧密协同工作的系统的部件，Impala 和存储引擎分离出来。在下图中，表示 Impala 的高层次架构。



一个完整地 Impala 系统由三个服务所组成的。首先 Impala 的守护进行也被称为 Impalad 服务负责从客户端进程中接收查询请求，并将查询任务分布到簇中。除次之外，Impalad 还能够为其它 Impalad 进程执行单一的查询片段。当一个 Impalad 作为领导者执行查询操作时，它就被称为那个查询的协调者。然而所有的 Impalad 进程都是相等地位的，它们能够扮演所有的角色。Impala 的这一特点有利于容错和负载均衡。

Impala 的守护进程被部署在所有的主机上，而这些主机也同时运行了数据节点进程。Impala 的这个特点允许它能够利用数据本地化，即能够读取文件系统的数据而不用使用网络。

Statestored 进程是 Impala 的元数据发布-订购服务，它能够将簇内的元数据发送到所有的 Impala 进程。在整个 Impala 系统中一般只会有一个单独的 Statestored 实例。

最后，Catalogd 进程，主要职能是 Impala Catalog 的持续化存储库和元数据访问路由。通过 Catalogd 进程，Impalad 在执行 DDL 命令的同时能够

将其在外部 catalog 中进行映射，例如 Hive 的 Matastore。对于提供 catalog 的改变通过 statestore 进行广播。

所有这些 Impala 服务，包括几个设置选项，例如资源池的大小，系统内存等，都被 Cloudera Manager 控件所管理。Cloudera Manager 是一个簇管理应用。它不仅能够管理 Impala，并能够管理其他很多的 Hadoop 部署服务。

状态分布

在很多主机上运行的 MPP 数据库设计中一个主要的挑战是怎样能够做到簇宽元数据的协调和同步。Impala 的均衡节点结构要求所有节点必须能够接受和执行查询操作。因此，所有节点必须拥有实时的系统 catalog 和最近的 Impala 簇成员信息以便于查询操作能够被正确地安排。

我们也许能够通过部署一个分布式的簇管理服务和所有簇范围的元数据的标准版本号。Impala 的守护进程能够以懒策略来查询这部分数据，这保证了所有的查询都返回了实时的响应。然而，在 Impala 设计中一个基本原则是避免同步 RPC。不管这些小号，我们发现查询延迟通常被建立 TCP 连接的时间和加载一些远程服务的时间锁拖慢。相应地，我们设计了 Impala 来推动部分的更新，并设计了一个简单地发布-订阅服务，被叫做 Statestore 的来传播元数据变化到所有订阅者中。

Statestore 保留了一系列的 Topics。Topics 指的是（键，值，版本）对的数组，这些对也被叫做入口，其中的键和值是字符类型的数组，而版本是

64 位的整数。Topic 被应用定义，所以 Statestore 并不了解 topic 入口内容的意义。Topics 在 Statestore 的生命周期中是持续的，但是在服务重启后即丢失了。希望接收到更新的进程被称作订阅者。它们在一开始就向 Statestore 注册并提供 Topics 的列表。Statestore 进程通过向注册者针对每一个被注册的 Topic 发送初始化的 Topic 更新，这个初始化的 Topic 包含了所有在那个 Topic 中包含的入口。

在注册之后，Statestore 向订阅者发送两种类型的消息。第一种消息是 Topic 的更新，并且包含了在上一次成功向订阅者发送 Topic 更新后所有的变化。

每一个订阅者都维护了一个最近版本号，能够允许 Statestore 能够只发送更新之间的差值。响应与一个 topic 的更新，每一个订阅者发送变化列表。这些改变被确保在下次更新被收到之前已经被应用。

第二种类型的 Statestore 消息是 Keepalive，即存活消息。Statestore 使用 Keepalive 消息来维持与每一个 Subscriber 之间的联系，否则订阅者将会认为订阅超时并试图重新订阅。之前的 Statestore 版本使用 Topic 更新消息来达到 Keepalive 的目的，但是由于 topic 更新数量的增长，定时地向每一台订阅者发送更新变得困难，导致了订阅者失败检测进程错误地响应。

如果 Statestore 检测到了一个失败的发布者（例如，重复发送 Keepalive 信号失败），这将会终止发送更新。一些 Topic 入口也许会被标记为短暂的，表示如果他们拥有的订阅者发生错误，他们也就会被移除。

Statestore 提供非常弱的语义：Subscribers 也许会被以不同的速率更新（虽然 Statestore 进程总是试图将更新公平地分布到各个节点），因此可能

会产生多个不同的 topic 内容的版本。然而，Impala 仅仅使用 topic 的元数据在本地做出决策，而不是通过簇内节点的协同合作。例如，查询计划根据 catalog 的元数据 Topic 在单一节点上得到执行。一旦一个完整的计划被计算，所有那个计划执行所需要的信息被直接分布到各个执行节点。一个执行节点不需要获知元数据主题的相同版本。

虽然在整个 Impala 部署中只有一个 Statestore 进程，我们发现它能够很好地适应于中等规模的部署。通过相同的配置，也能够很好地服务于大型的部署。Statestore 不将任何元数据存到磁盘，所有当前的元数据被各个活动 Subscribers 推送到 Statestore，例如载入信息。因此，如果一个 Statestore 被重启，它的状态也能够通过初始化的 Subscriber 注册过程被恢复。或者如果一台主机上，Statestore 进程运行失败了，一个新的 Statesotre 进程能够在任何节点被重启，并且一个 Subscribers 也可能订阅失败。在 Impala 中没有内嵌的失败克服机制，取而代之，部署的时候使用了一个重启 DNS 入口来强迫 Subscriber 去自动移动到新的 Statestore 进程实例。

Catalog 服务

Impala 的 catalog 服务通过 Statestore 的广播机制为 Impala 的守护进程提供元数据。另外，它还为 Impala 守护进程执行 DDL 操作。Catalog 服务从第三方元数据存储中获得信息，例如 Hive 的元数据获得 HDFS 的名字节点，并且将那些信息转换到 Impala 兼容的 catalog 结构。这个结构允许 Impala 对于它所依赖的元数据仓库引擎是关系不可知的，而这一特点允许我们将快速地将新的元数据仓库添加到 Impala 关系中，例如 HBase 支持。

所有系统 Catalog 的变化通过 Statestore 分发。

Catalog 服务也允许我们使用 Impala 定义的信息来反对系统的 Catalog。例如，我们仅仅使用 Catalog 服务来注册用户定义功能信息（不将这些发布到 Hive 的元数据中）因为它们是专门用于 Impala 的。

因为 Catalogs 通常很大，并且访问的表很少是均匀的，所以 Catalog 服务仅仅为每张表载入了一个框架性的入口。更加详细的信息能够在后台慢慢从第三方数据仓库中被载入。如果一张表在它完全载入之前被请求到，一个 Impala 的进程将会检测并且将向 catalog 服务的请求置于优先地位。这一点将会阻塞请求直到整张表被载入。

4. 前端

Impala 的前端负责编译 SQL 文本，将其转换为能够被后端执行的查询计划。前端通过 Java 语言写成，包含了一个 SQL 语句分析器和基于成本的查询优化器，而这些都是从头开始写的。在基本的 SQL 特征的基础上，包括了 select，project，join，group by，order by， limit， Impala 支持 Inline views、不相关和相关子查询（被作为 joins 操作重写），所有的外连接的变种包括明确左/右半和反连接，和窗口分析功能。

查询编译进程遵守传统的工作分工：查询分析，语义分析，和查询计划/优化。我们将会在后面的篇幅中关注在查询编译中最具挑战性的部分。对于查询计划的执行中包括了两个阶段：单点查询和查询并行和分片。

在第一阶段，查询树被转换为一个不可执行的单点计划树，包含了下面的查询节点：HDFS/HBase 扫描，哈希连接，交叉连接，组合，哈希聚

合，排序，和分析估计。这一步骤负责为在低层次可能的计划节点负责断言。成本分析被寄予 table/分块基数加上每一列的不同值的总数。

计划的第二个阶段将单节点计划作为输入并输出一个分布式的执行计划。总体的目标是数据迁移最小化，将扫描区域最大化。在 HDFS 中，远程读取要比本地读取慢得多。计划通过在必要的时候在计划节点之间增加交换节点来分布。在必要地时候通过增加额外的非交换计划节点来最小化网络数据迁移。在这第二个阶段，我们为每一个连接节点（在这一点中连接顺序是固定的）决定连接策略。所支持的连接策略是广播和分片。Impala 选择能够最小化网络数据交换的策略，并利用存在的数据分片作为连接输入。

5. 后端

Impala 的后端从前端接收查询片段并负责它们的执行。它被设计于能够利用现代硬件。后端使用 C++写成并在执行阶段使用代码生成来产生高效率的代码路径和最小化的存储开销。

Impala 促使数十年在并行化数据库上的研究发生改变。执行模型是传统的 Volcano-style 加上交换执行器。处理以批处理的方式被执行，每个 GetNext 的调用对行执行了批处理。得益于 stop-and-go 异常执行器，执行能够被流水化，这使存储最小化来存储立即结果。

那些可能会消耗大量数据的执行器被设计成能够将它们的工作的一部分数据保存到磁盘在需要的情况下。能够被泄露的操作包括了哈希连接，基于哈希的断言，排序和分析功能估值。

Impala 采用了一种分片的方法来执行哈希连接和断言操作。换言之，每一个元组的哈希值的一些位决定了目标，而剩下的位则决定了哈希表的探查。当所有的哈希表能够适应于内存时，分片步骤的开销是最小的，能够比非泄露，非基于分片实现降低 10%的开销。当有内存压力时，一个受害者分片可能会被移到磁盘中，通过这种方法来为其它分片腾出空间而让它们能够完成他们的执行过程。当监理一个哈希表来做哈希连接操作，我们创建一个 Bloom 过滤器，这个过滤器接下来被传递到探索阶段扫描器，实现了一个简单版本的半连接。

6. 结论

在本论文中我们介绍了 Cloudera Impala，这是一个开源的 SQL 引擎，能够将并行 DBMS 技术带到 Hadoop 环境。我们的性能结果表明除开 Hadoop 作为一个批处理环境的事实，监理一个分析型的 DBMS 在 Hadoop 的顶层是可行的。并且它在性能上能够表现地和目前的商业处理方法一样好。但在同时，它也保留了 Hadoop 的灵活性和成本优势。

目前，Impala 已经能够在很多的工作领域代替传统的单片分析性关系型数据库系统。我们能够预测随着时间的推移，在这些系统间在 SQL 功能上的差距将会渐渐消失。然而，我们认为 Hadoop 环境中的模块化现状有一些优势是传统的大片分析型关系型数据库管理系统所不能实现的。特别的，将文件形式和处理框架混合的能力代表了很多的任务能够被单一的系统执行，而不需要数据迁移。

在 Hadoop 生态圈中得数据管理缺乏了一些功能，而这些功能已经在过

去的年代中被传统的关系型数据库管理系统所实现了。除开这些，我们希望上文中提到的这两种数据库之间的性能差距能够快速缩小，然后这种开放式模块化环境能够允许它在不久的将来成为主要的数据管理结构。