北京郵電大學

本科毕业设计（论文）

题目: 海量数据的实时性查询处理研究

姓　　名 李斌

学　　院 电子工程学院

专　　业 电子信息科学与技术

学　　号 08210997

班内序号 12 或 13

指导教师 侯宾

2012 年　　6 月

北　京　邮　电　大　学

本科毕业设计 (论文) 诚信声明

本人声明所呈交的毕业设计 (论文)，题目《海量数据的实时性处理查询》是本人在指导教师的指导下，独立进行研究工作所取得的成果。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京邮电大学或其他教育机构的学位或证书而使用过的材料。

申请学位论文与资料若有不实之处，本人承担一切相关责任。

本人签名：_____　日期：_____

海量数据实时性查询研究

摘　要

为了应对互联网上爆发性的信息量的存储和查询要求，目前有许多种的解决方案。传统的方法使用 RDBMS 关系型数据库如 mysql 及它的查询引擎来处理，但这需要维护非常多的表间约束，以及它不能灵活的扩展等。Hadoop 是一个开源的分布式架构，在其之上的项目 hive 提供了功能强大的引擎来和 Hadoop 分布式优势相结合。HBase 和 MongoDB 都是典型的非关系型数据库，其中 HBase 面向列族，MongoDB 功能强大面向文档，他们都提供了独特的数据库存储模式。

本论文采用了横向比较实时性信息的方法，主要关注于查询的时间消耗，总共安排了大小数量即简单复杂查询共四种情况进行对比。经过具体的实际测试，发现 mysql 在中小数量级上的查询性能有明显的优势，只是在非常大的数量上的复杂查询性能不如 hive. 由于 HBase 本身不能支持条件查询，最初考虑采用 hive 和 hbase 整合的方法，但结果发现并不理想。

为了提高实时性，本论文挑选了几个非常重要的方面进行讨论。如采用索引，调整参数等。并在最后对连接查询的几个主流实现算法进行了研究。

HBase 在官网里提供了 api 使得客户端能够通过编写 java 程序来操作 Hbase 中存在的数据。本文最后部分参考了官网的手册学习了几个重要概念并且针对前面做过的实验写出了相应的程序并成功得到了结果。

关键字　Hive　HBase　实时性查询

research on query of huge amount of data

ABSTRACT

In order to meet the demand of storing and querying explosive huge amount of data produced by Internet.There has been many solutions. Traditional solutions use RDBMS such as mysql and its query engine, but it requires to maintain a lot of constrains between tables in mysql.Hadoop is a distributed open source architecture, and hive is a project based on Hadoop providing powerful engine with Hadoop's advantage.HBase and MongoDB are typical non-relational database.Hbase is column-oriented and MongoDB is document-oriented.Both of them provide unique storage pattern.

In this thesis, we adapt horizontal comparison of real-time performance method, and it focuses on the query time consumption. We arrange for a number of experiments range from simple ones with little amount to complex ones with large amount.After the specific pratical test, I found mysql big advantage over others on small and medium-sized query, while hive is supposed to be better on large data and complex queries. I initially consider using integration of hive and hbase to solve the problem that hbase doesn't support conditional query,but in the end ,the result is not satisfactory.

To improve the real-time performance, this paper selects a few important aspects to discuss,such as the use of index ,adjusting parameters and so on. The final part talks about several mainstream algorithm.

The official webiste of Hbase provides api that allows us to write java programs to manipulate data in Hbase. The last part of this article study a few concepts first and i write a corresponding program for experiments earlier and get successful results.

KEY WORDS Hive HBase real-time query

目　　录

第一章　概述 …… 1
1.1　背景介绍 …… 1
1.1.1　大量的数据 …… 1
1.1.2　传统方法存在的问题 …… 1
1.1.3　Hadoop 架构 …… 2
1.2　一些解决方案 …… 3
1.2.1　RDBMS …… 3
1.2.2　Hive …… 3
1.2.3　HBase …… 4
1.2.4　MongoDB …… 6
1.3　论文结构 …… 6
第二章　研究方法 …… 7
2.1　Hive 和 HBase 的整合 …… 7
2.1.1　原理 …… 7
2.1.2　整合的过程 …… 7
2.2　数据导入 …… 10
2.2.1　将数据导入到 hive …… 10
2.2.2　将数据导入到 hive/hbase 中 …… 11
2.2.3　导入数据到 mongoDB …… 14
2.2.4　导入数据到 mysql 中 …… 15
第三章　查询实验结果 …… 16
3.1　简单查询 …… 16
3.1.1　小数量级 …… 16
3.1.2　大数量级 …… 18
3.2　复杂查询 …… 22
3.2.1　小数量级 …… 22
3.2.2　大数量级 …… 23
3.3　总结 …… 25
第四章　查询优化 …… 26
4.1　使用索引 …… 26
4.2　参数优化 …… 27
4.2.1　如何设定参数 …… 27
4.2.2　列裁剪（Column Pruning） …… 27
4.2.3　分区裁剪（Partition Pruning） …… 27
4.2.4　Join 和 MapJoin …… 28
4.2.5　groupby …… 28

4.2.6 合并小文件 …… 28
4.3 解决数据倾斜 …… 28
4.4 对 join 查询算法的研究 …… 29
第五章 HBase 客户端编程 …… 32
5.1 客户端编程的重要概念 …… 32
5.1.1 HBaseConfiguration …… 32
5.1.2 HBaseAdmin …… 32
5.1.3 Put 和 Get/Scan …… 33
5.1.4 Filters …… 33
5.2 应用实例 …… 33
5.2.1 代码 …… 33
5.2.2 编译并打包 …… 35
5.2.3 执行 …… 35
参考文献 …… 36
致谢 …… 38

第一章　概述

本章主要介绍课题研究的相关背景，以及主要用到的开源软件的特点和基本概念。

1.1　背景介绍

1.1.1　大量的数据

当前，随着整个互联网的快速发展，海量的数据每天都在无时不刻的产生。一方面，有类似 facebook、twitter 等社交网站如雨后春笋般的涌现，每个用户都要产生大量的动态信息，包括用户进行发布个人状态，日志等资料，以及在网站上的点击，收听音乐等操作，还有一些文档抓取等等操作，事实上，facebook 每天都要产生超过 2TB 的数据量，而这还是经过了科学的压缩之后得到的。除了社交网络之外，电子商务网站也需要产生大量的日志用来记录用户的行为。另一方面，从中小型到大型的数据密集型企业，如电信，金融，政府，零售等等也需要保存几十个到上百个 TB 的用户数据。

根据[1] 的介绍：

- 纽约证券交易所每天产生 1TB 的交易数据
- 著名社交网站 Facebook 的主机存储着约 100 亿张照片，占据 PB 级存储空间
- 互联网档案馆存储着约 2PB 数据，并以每月至少 20TB 的速度增长
- 瑞士日内瓦附近的大型强子对撞机每年产生约 15PB 的数据

这么多的数据带来的问题就是如何进行存储，如何在这些数据之上进行查询以发掘有效信息，如何在查询时保证良好的实时性。当然，要想处理巨大数据量的计算一定要用到分布处理技术，然而怎么处理，怎么并发，尚且有很多的内容值得研究。

1.1.2　传统方法存在的问题

在数据量很小的时候，大都是将数据部署在一个服务器上，并将数据存储在关系型数据库 RDBMS 中。然而，这种方法在超大规模和高并发的需求面前显得力不从心，这主要体现在：

- 对数据库高并发读写的需求

  web2.0 网站要根据用户个性化信息来实时生成动态页面和提供动态信息，所以基本上无法使用动态页面静态化技术，因此数据库并发负载非常高，往往要达到每秒上万次读写请求。关系数据库应付上万次 SQL 查询还勉强顶得住，但是应付上万次 SQL 写数据请求，硬盘 IO 就已经无法承受了。其实对于普通的 BBS 网站，往往也存在对高并发写请求的需求，例如像 JavaEye 网站的实时统计在线用户状态，记录热门帖子的点击次数，投票计数等，因此这是一个相当普遍的需求。

- 对海量数据的高效率存储和访问的需求

  以 Friendfeed 为例，一个月就达到了 2.5 亿条用户动态，对于关系数据库来说，在一张 2.5 亿条记录的表里面进行 SQL 查询，效率是极其低下乃至不可忍受的。再例如大型 web 网站的用户登录系统，例如腾讯，盛大，动辄数以亿计的帐号，关系数据库也很难应付。

- 对数据库的高可扩展性和高可用性的需求

  在基于 web 的架构当中，数据库是最难进行横向扩展的，当一个应用系统的用户量和访问量与日俱增的时候，你的数据库却没有办法像 web server 和 app server 那样简单的通过添加更多的硬件和服务节点来扩展性能和负载能力。对于很多需要提供 24 小时不间断服务的网站来说，对数据库系统进行升级和扩展是非常痛苦的事情，往往需要停机维护和数据迁移，为什么数据库不能通过不断的添加服务器节点来实现扩展呢？

此外，RDBMS 的优势如数据库的事务一致性、多表关联复杂查询等需求并不是很强烈，因而它的优势得到了削弱。[2] 指出了 RDBMS 和 NOSQL 之间的优劣关系。

1.1.3 Hadoop 架构

hadoop 是一个能够很好解决上文提到的海量数据的存储需求。事实上，作为一个分布式处理框架，它拥有者许多可贵的优点：比如高容错性、易于扩展、能够部署在十分廉价的硬件上。此外，他还能通过维护多个副本来保证可靠性。

Hadoop 包括两个部分，其一是 Hadoop 分布式文件系统 HDFS，它很适合哪些有大数据集的应用，并提供了对数据读写的高吞吐率。HDFS 是一个 master/slave 的结构，就通常的部署来说，在 master 上只运行一个 NameNode，而在每一个 slave 上运行一个 Datanode。同时它也支持传统的层次文件组织结构，同现有的一些文件系统在操作上很类似，如创建和删除文件及文件夹等等。Hadoop 假设硬件出错是一种正常的情况，而不是当作小概率突发事件看待，HDFS 设计的目标是在成百上千台服务器中存储数据，实时高效地检测出硬件错误，并且快速进行完整性补救或数据自动恢复。

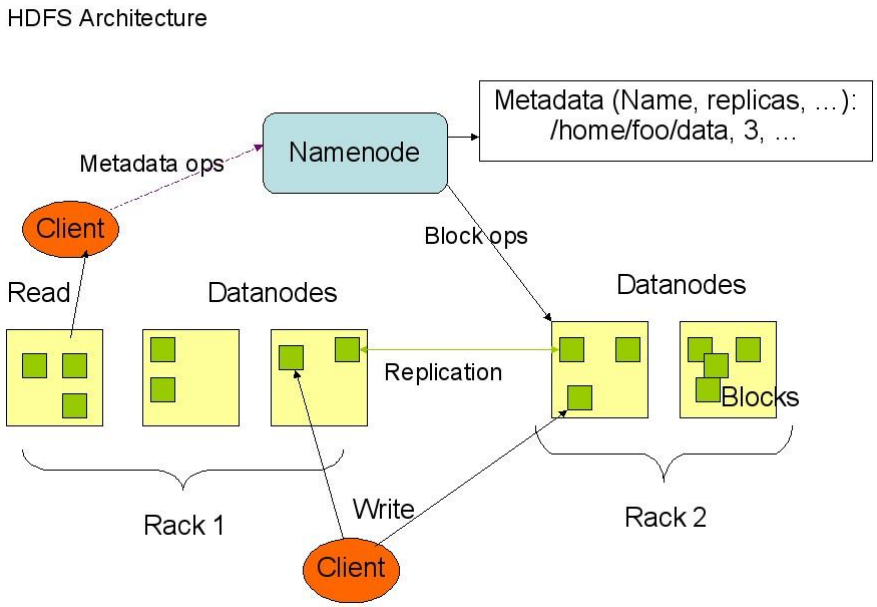


图 1-1　HDFS 架构

第二个部分是 MapReduce 编程模型。它将并行计算性、容错性、负载均衡、数据分布方式等复杂的细节都封装在一个库里面，用户只要直接表述想要执行的简单

运算即可。设计与实现 Map/Reduce 框架模型的突出优点是通过简单的接口来实现自动的并行化和大规模的分布式计算，并实现在大量的普通 PC 机上实施高性能计算。MapReduce 将数据计算划分为两个步骤: 分割计算 (MaP) 与合并统计 (尺 educe)，这是其达到并行计算的主要工作方式。

Hadoop 技术已经在互联网领域得以广泛的应用。如 Yahoo！使用 Hadoop 技术进行广告系统和 Web 搜索；Facebook 使用 Hadoop 存储日志数据及之上的数据分析和机器学习；百度使用 Hadoop 进行搜索日志分析和网页数据挖掘工作以及淘宝使用 Hadoop 系统用来存储并处理电子商务的交易相关数据。

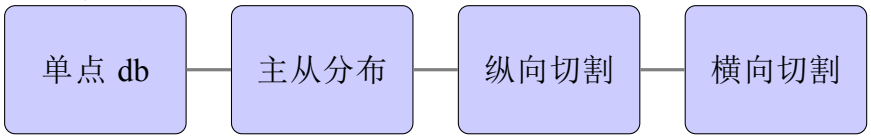
1.2 一些解决方案

为了实现大量数据的实时性查询，主要有以下几种方案：

1.2.1 RDBMS

典型的代表就是 Mysql。它的优点有：模式固定，具有 ACID 性质和复杂的 SQL 查询处理引擎，能够保证数据的一致性和完整性。但另一方面，他也有许多难以避免的缺点，主要有：在上亿条记录里进行查询效率十分低下，无法进行简单的扩展。

为了将 Mysql 满足海量数据查询的需要，就必须进行扩展为关系型数据库集群。当前的思路是随着数据量的增加，使得单点的结构变为主从型的分布，这样就能在从节点进行读操作，缓解了读操作时造成的 IO 压力。另外，如果数据库中的表数目较多，则可以将不同的表分开存储；如果某一个表太大，则可以把表分为不同的分区同样进行分布式存储。



1.2.2 Hive

Hive 是建立在 Hadoop 上的数据仓库基础构架。它提供了一系列的工具，可以用来进行数据提取转化加载（ETL），这是一种可以存储、查询和分析存储在 Hadoop 中的大规模数据的机制。Hive 定义了简单的类 SQL 查询语言，称为 QL，它允许熟悉 SQL 的用户查询数据。同时，这个语言也允许熟悉 MapReduce 开发者的开发自定义的 mapper 和 reducer 来处理内建的 mapper 和 reducer 无法完成的复杂的分析工作。Hive 没有专门的数据格式。Hive 可以很好的工作在 Thriff 之上，控制分隔符，也允许用户指定数据格式。[3][4] 介绍了 Hive 的特点和日志分析的应用。

下图是 hive 的架构:

Hive 的用户接口包括 CLI，Client 和 WUI。其中最常用的是 CLI，Cli 启动的时候，会同时启动一个 Hive 副本。Client 是 Hive 的客户端，用户连接至 HiveServer。在启动 Client 模式的时候，需要指出 HiveServer 所在节点，并且在该节点启动 HiveServer。WUI 是通过浏览器访问 Hive。

Hive 将元数据存储在数据库中，如 mysql、derby。Hive 中的元数据包括表的名字，表的列和分区及其属性，表的属性（是否为外部表等），表的数据所在目录等。

解释器、编译器、优化器完成 HQL 查询语句从词法分析、语法分析、编译、优化

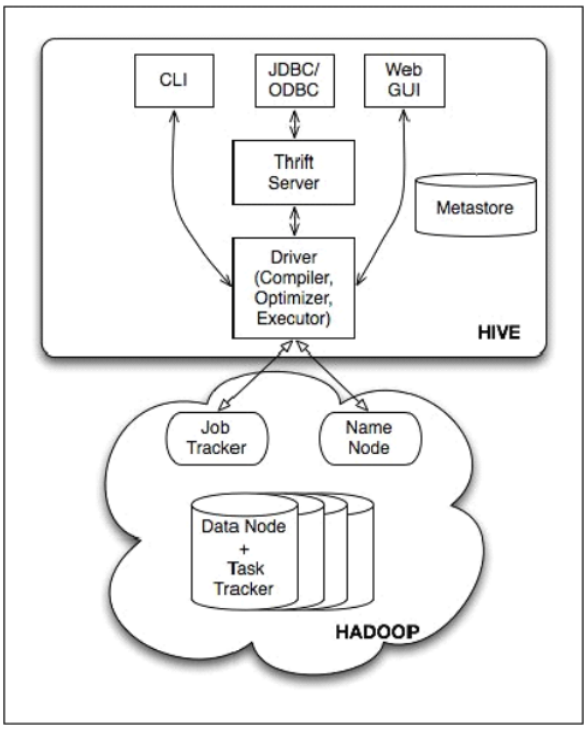


图 1-2　Hive 的架构图

以及查询计划的生成。

生成的查询计划存储在 HDFS 中，并在随后有 MapReduce 调用执行。Hive 的数据存储在 HDFS 中，大部分的查询由 MapReduce 完成。

1.2.3　HBase

HBase 是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统，利用 HBase 技术可在廉价 PC Server 上搭建起大规模结构化存储集群。

HBase 是 Google Bigtable 的开源实现，类似 Google Bigtable 利用 GFS 作为其文件存储系统，HBase 利用 Hadoop HDFS 作为其文件存储系统；Google 运行 MapReduce 来处理 Bigtable 中的海量数据，HBase 同样利用 Hadoop MapReduce 来处理 HBase 中的海量数据；Google Bigtable 利用 Chubby 作为协同服务，HBase 利用 Zookeeper 作为对应。

HBase 的服务器体系结构也是遵从简单的主从服务器架构，由 Hregion 服务器群和 HBase Master 主服务器构成。一般用表名 + 开始/结束主键来区分一个 Hregion，一个 Hregion 会保存一个表里面某段连续的数据。一般一台机器上面运行一个 Hregion 服务器，而每一个区段 Hregion 只会被一个 Hregion 服务器维护。也就是说，HRegion 是 HBase 中分布式存储和负载均衡的最小单元。但它并不是最小的存储单元。事实上，HRegion 由一个或多个 Store 组成，每个 store 又由一个 memstore 和 0 至多个 StoreFile 组成。每个 store 保存一个 column family。StoreFile 以 HFile 的格式保存在 HDFS 上。

下图是 HBase 的架构图：

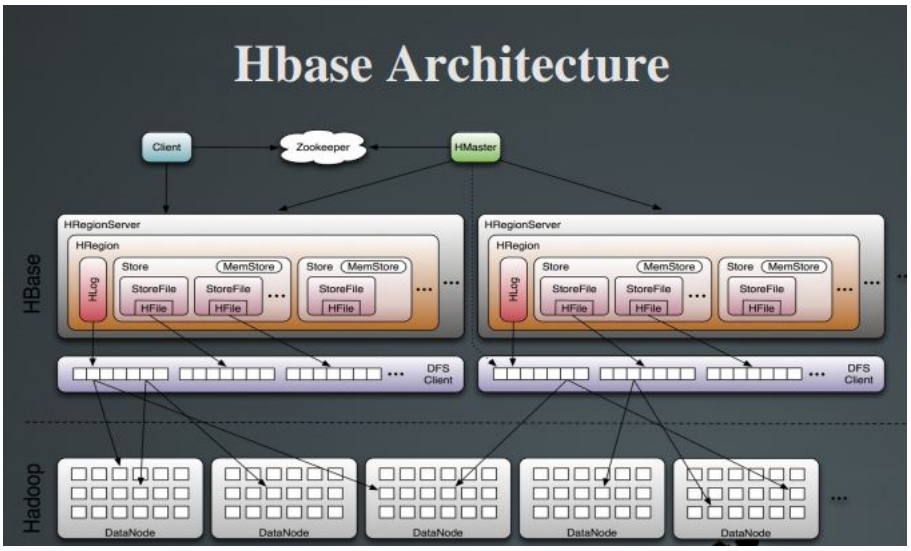


图 1-3　HBase 的架构图

HBase 的表由行和列组成，列划分为若干个列族。表中的 row key 是用来检索记录的主键，访问表中的行可以通过单个 row key 访问，也可以通过 row key 的 range，还有就是全表扫描。行的一次读写是原子操作。

Hbase 表中的每一列都归属于某个列族，列族是表的 chema 的一部分 (而列不是)，必须在使用表之前定义，列名都以列族作为前缀。访问控制、磁盘和内存的使用统计都是在列族层面进行的。HBase 中的表一般特点为大，甚至可以达到上百万行以至于上亿行；面向列，列族独立索引；稀疏，对于空的列，并不占用存储空间。下面是一个表的示例：

Row Key	column-family1		column-family2			column-family3
	column1	column1	column1	column2	column3	column1
key1	t1:abc t2:gdxdf		t4:dfads t3:hello t2:world			
key2	t3:abc t1:gdxdf		t4:dfads t3:hello		t2:dfdsfa t3:dfdf	
key3		t2:dfadfasd t1:dfdasddsf				t2:dfxxdfasd t1:taobao.com

图 1-4　HBase 的表结构

从上面的表结构可以看出，Hbase 的列不仅可以为空，亦可以动态进行增加，这就相对比较灵活。此外，Hbase 的存储是自动分片的，少了人工操作的麻烦。它的缺点是没有一个功能强大的查询引擎，不支持复杂查询。

1.2.4 MongoDB

MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。他支持的数据结构非常松散，是类似 json 的 bjson 格式，因此可以存储比较复杂的数据类型。Mongo 最大的特点是他支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。[5] 这本书比较详细的介绍了 MongoDB 的特性。

它的特点是高性能、易部署、易使用，存储数据非常方便。它扩展了关系型数据库的众多有用功能，如辅助索引、范围查询和排序，并且内置对 MapReduce 式聚合的支持，以及对地理空间索引的支持。它的数据模型对于开发者来说非常友好，配置选项也很轻松，并且有驱动程序和数据库 shell 提供的自然语言式放入 API。MongoDB 主要的功能特性有：

- 面向集合存储，易存储对象类型的数据。
- 模式自由。
- 支持动态查询。
- 支持完全索引，包含内部对象。
- 支持查询。
- 支持复制和故障恢复。
- 使用高效的二进制数据存储，包括大型对象（如视频等）。
- 自动处理碎片，以支持云计算层次的扩展性。
- 支持 RUBY，PYTHON，JAVA，C++，PHP 等多种语言。
- 文件存储格式为 BSON（一种 JSON 的扩展）。
- 可通过网络访问。

1.3 论文结构

下面是这篇文章整体的布局情况。

第一章　阐述选题的背景以及主要用到的几个开源软件的特点介绍。

第二章　阐述课题研究的方法；将 hive 与 hbase 的结合原理；实验的前提工作数据导入等等。

第三章　主要介绍了测试用到的几个实验，实验过程，实验结果及结论。

第四章　讨论了优化的方法如调整参数，使用索引，保证不倾斜；讨论了 join 算法。

第五章　介绍了 HBase 客户端编程的概念；针对一个条件查询编写了相应程序。

第二章　研究方法

本课题主要目的是为了着重在实时性方面进行主要的研究。所以总的研究采用了横向比较的思路，即在各个不同的解决方案如 RDBMS 和 NOSQL 之间进行控制变量的比较。整个比较的过程分为基础性的调研，并在实验的基础上得出结论。而后根据实验的结果进行实时性上的优化。采取各种各样的方法力图将每一个查询处理所用到的延迟时间降到最低，以期满足性能上的要求。

基础性调研具体来说就是将相同的查询应用到同一数据上进行对比。这里的查询包括常见的简单查询和复杂查询，如连接查询；这里的数据分为小数量级和大数量级；这里的比较内容主要是查询所消耗的时间，即实时性指标；这里的比较是在 mysql、hive、hbase 和 mongoDB 之间。

改进的过程具体来说就是充分发挥各个开源产品的优点，能够分布式的进行分布式查询，能够将表结构进行优良设计的进行优良设计，能够修改相关参数的争取把参数调到最优，能够改进实现算法的改进实现算法。以完成查询处理的最佳实践。

2.1　Hive 和 HBase 的整合

由于非关系型数据库 hbase 不提供条件查询的接口，但是作为一个面向列簇的 NOSQL 数据库它又具有自己独特的优点。为了能够较为方便的实现 HBase 的条件查询，考虑到 hive 的查询引擎使用的方便性，我们这里利用到了 apache Hbase 官方网站[1]上提供的整合工具 hive-hbase-handler.jar 实现二者的整合。

2.1.1　原理

图 2-1 是这种方法的实现原理。

如图所示，左边的部分是 hive 的架构图，之前的 hive 是直接建立在 Hadoop 集群之上的，也就是说 hive 作为一个查询引擎驱动直接操作 HDFS 中存放的数据。而现在可以将 Hive 作为 Hbase 的 client，这样一来就能够通过 hive 来实现数据的条件查询，复杂查询等等，另一方面数据还是存在 HBase 中的。总体的效果就是 hive 中插入的数据无论在 hive 还是在 hbase 中都能看到而且效果一样。同理，在 hbase 中修改数据后在 Hive 中查询得到的结果也会随之改变。

2.1.2　整合的过程

1. 将 hbase-0.90.5.jar 和 zookeeper-3.3.2.jar 拷贝到 hive/lib 下。注意：如何 hive/lib 下已经存在这两个文件的其他版本（例如 zookeeper-3.3.1.jar），建议删除后使用 hbase 下的相关版本。

   拷贝 hbase-site.xml 到 hive/conf 目录下 (我在没拷贝时出现过 hbase master not running exception，拷贝后就好了)。

   拷贝 hbase/conf 下的 hbase-site.xml 文件到所有 hadoop 节点 (包括 master) 的 hadoop/-conf 下。

[1] http://wiki.apache.org/hadoop/Hive/HBaseIntegration

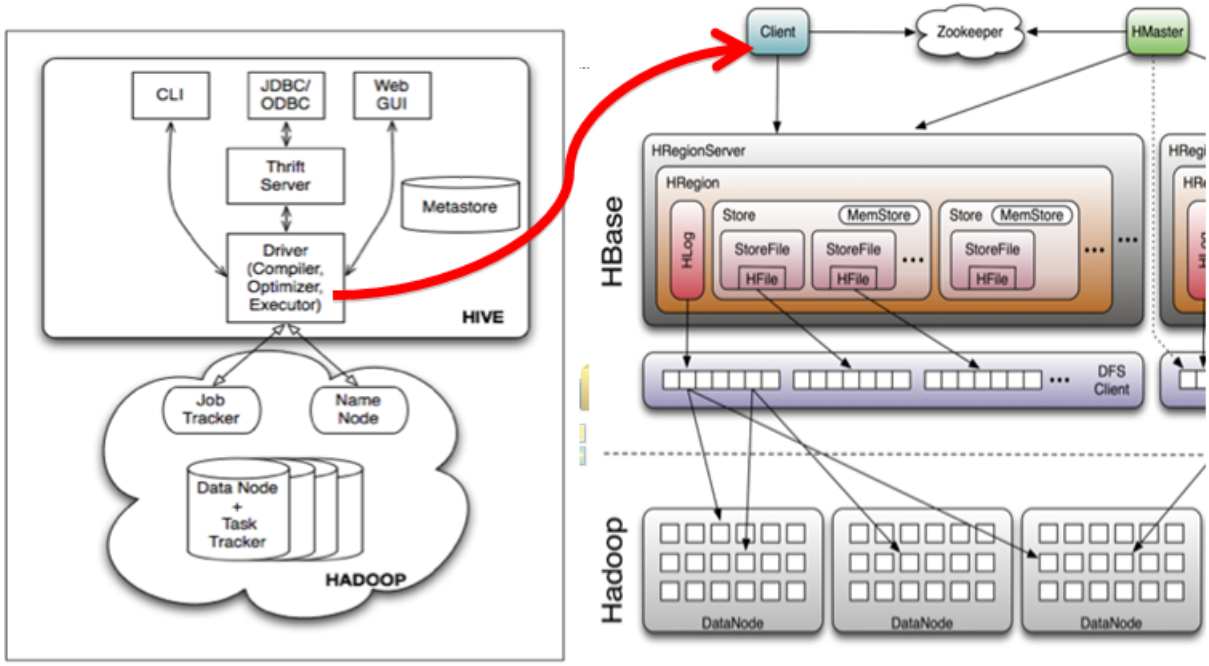


图 2-1　Hive 和 HBase 整合的原理

这一部分就是通过拷贝配置文件使得 hive 和 hbase 互相了解对方的版本和具体的配置细节，以便于 handler 调用。

2. 在 $HIVE_HOME/conf 目录中修改文件 hive-site.xml:

```
<configuration>

<property>

   <name>hive.aux.jars.path</name>

   <value>file:///home/cubika/hadoop/hive-0.8.1/lib/hive-hbase-handler-0.8.1.jar
,file:///home/cubika/hadoop/hive-0.8.1/lib/hbase-0.90.5.jar,file:///home/cubika/
hadoop/hive-0.8.1/lib/zookeeper-3.3.4.jar</value>

</property>

</configuration>
```

图 2-2　hive-site.xml 的设置

如图所示，这个配置文件的意思就是在启动时指定了各个软件的 jar 包的路径，这样启动后的 handler 就可以调用相应的 jar 包。

3. 单节点启动方法：bin/hive –hiveconf hbase.master=master:60000 集群启动方法：bin/hive –hiveconf hbase.zookeeper.quorum=slave

4. 创建 hbase 识别的表:

```
CREATE TABLE hbase_table_1(key int, value string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,cf1:val")
```

```
TBLPROPERTIES ("hbase.table.name" = "xyz");
```

创建表时需要注意到的两点是：1. 存储时用到了 HBaseStorageHandler，这个工具的作用就是可以用它来创建新的 HBase 表格、删除 HBase 中的表格等等。2. 指定时加入映射的关系，即 hbase.columns.mapping，将 hive 表中的某一列映射为 hbase 的某一列族的某一列，在指定时必须要明确用：key 来使某一列称为 Hbase 的 rowkey，另外映射前后表的列数必须相同。

5. 导入数据

```
hive> CREATE TABLE pokes (foo INT, bar STRING);
hive> LOAD DATA LOCAL INPATH './examples/files/kv1.txt' OVERWRITE INTO
    TABLE pokes;
hive> INSERT OVERWRITE TABLE hbase_table_1 SELECT * FROM pokes WHERE
    foo=86;
```

这样就在整合后的表 hbase_table_1 中插入了一条记录 foo=86。

6. 分别在 hive 中和 hbase 中查看数据

在 hive 中查看：

```
hive> select * from  hbase_table_1;
```

```
hive> select * from hbase_table_1;
OK
86      val_86
Time taken: 1.112 seconds
```

在 hbase 中查看：

```
hbase(main):001:0> describe 'xyz'
hbase(main):002:0> scan 'xyz'
```

```
hbase(main):001:0> describe 'xyz'
DESCRIPTION                                          ENABLED
 {NAME => 'xyz', FAMILIES => [{NAME => 'cf1', BLOOMF true
 ILTER => 'NONE', REPLICATION_SCOPE => '0', COMPRESS
 ION => 'NONE', VERSIONS => '3', TTL => '2147483647'
 , BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCK
 CACHE => 'true'}]}
1 row(s) in 4.1410 seconds

hbase(main):002:0> scan 'xyz'
ROW                    COLUMN+CELL
 86                    column=cf1:val, timestamp=1331710202463, value=val_86
1 row(s) in 2.2150 seconds
```

从上两幅图中可以看出，无论对于 hive 中存在的表 hbase_table_1 还是 hbase 中的表 xyz, 他们两个已经息息相关，他们两个表中的数据是一样的，而且对任何一个表进行插入操作都会引起两个表的查询结果同时改变，这说明二者已经紧密结合。

2.2 数据导入

在基础性调研之前首先需要做的工作是进行各个表格的创建以及数据的导入。数据的导入方法随着数据库的不同而不同，但基本上大同小异。由于导入时数据量可以很大，导入的时间消耗其实也可以作为一项实时性的性能指标。

2.2.1 将数据导入到 hive

这里导入的文件格式是 csv – 逗号分隔型取值格式（英文全称为 Comma Separated Values，简称 CSV）。它是一种纯文本格式，用来存储数据。在 CSV 中，数据的字段由逗号分开，程序通过读取文件重新创建正确的字段。hive 中对于不同的格式有不同的导入方法，下面介绍 hive 导入 csv 格式文本文件的具体步骤。

1. 从 https://github.com/ogrodnek/csv-serde 下载 hive 导入 csv 的 serde 包, 在 hive 里将这个包的路径添加进来：add jar /home/cloud/csv-serde-1.1.2.jar。有了这个 jar 包才能进行 csv 格式文本的导入工作。

2. 创建表 bicc 这是一个电信的表格，每一项是一宗通话的各个具体的信息。建表语句为：

```
CREATE TABLE bicc
(
  START_TIME_S              string,
  START_TIME_NS             string,
  END_TIME_S                string,
  END_TIME_NS               string,
  CDR_INDEX                 string,
  SOURCE_IP                 string,
  DESTINATION_IP            string,
  CIC                       string,
  OPC                       string,
  DPC                       string,
  RELEASE_REASON            string,
  CALLING_NUMBER            string,
  CALLED_NUMBER             string,
  ORIGINAL_CALLED_NUMBER    string,
  TRANSFER_NUMBER           string,
  LOCATION_NUMBER           string,
  RESPONSE_TIME             string,
  ACM_TIME                  string,
  ANM_TIME                  string,
  REL_TIME                  string,
  CALL_DURATION             string,
  DURATION                  string,
  CODEC_MODIFY_FLAG         string,
  CODEC_MODIFY_RESULT       string,
  CODEC_NEGOTIATION_FLAG    string,
  CODEC_NEGOTIATION_RESULT  string,
  CODEC_TYPE                string,
  CALL_TYPE                 string,
  IS_EXT_PLATFORM           string,
  CALL_HOLD                 string,
  CALL_FORWARD              string,
  CALL_WAITING              string,
  CONFRENCE_CALL            string
```

```
)
row format serde 'com.bizo.hive.serde.csv.CSVSerde'
stored as textfile;
```

这里的 row format 其实是使用了刚刚添加的 jar 包 CSVSerde，目的就是告诉 hive 如何判断每一个域，具体来说就是以逗号作为每个域的分隔符，换行作为每个行的分隔符。

3. 插入数据：插入了 100 个记录，插入的源是文件 bicc_100.csv

```
bin/hadoop fs -put /home/cloud/data/bicc_100.csv /user/hive/warehouse/
    bicc
```

因为这里 hive 数据仓库在 HDFS 中的路径就是/user/hive/warehouse, 将文件直接放到这个目录下的 bicc 目录里其实也是相当于在这个表中插入了数据。另外用 hive 提供的 LOAD DATA INPATH 语法插入数据也可以。

同理可以新建两个大数据量的表格：yuyin(380 万个记录）和 duanxin（940 万个记录）。这两个表的结构是一样的，同样是："用户号码 | 漫游类型 | 区号呼叫发生地 | 呼叫日期 | 呼叫时间 | 长途类型 | 方向类型 | 通话时长 | 位置小区代码 | 蜂窝号基站代码 |"，建表的语句和上面类似，在这里就不在赘述了。

2.2.2 将数据导入到 hive/hbase 中

经过整合之后，hive/hbase 的所有操作和 hive 是完全一样的，唯一的不同就是需要首先启动 hbase 服务。因为只有开启了 hbase 服务后 handler 才能打开 hbase 中的表。下面是向整合后的表插入数据的各个具体步骤。

1. 创建 hive 和 hbase 整合后的表 SRC_TBD_BICC_CDR，这个表对应了上述的表 bicc。建表语句是：

```
create table SRC_TBD_BICC_CDR
(
  START_TIME_S              string,
  START_TIME_NS             string,
  END_TIME_S                string,
  END_TIME_NS               string,
  CDR_INDEX                 string,
  SOURCE_IP                 string,
  DESTINATION_IP            string,
  CIC                       string,
  OPC                       string,
  DPC                       string,
  RELEASE_REASON            string,
  CALLING_NUMBER            string,
  CALLED_NUMBER             string,
  ORIGINAL_CALLED_NUMBER    string,
  TRANSFER_NUMBER           string,
  LOCATION_NUMBER           string,
  RESPONSE_TIME             string,
  ACM_TIME                  string,
  ANM_TIME                  string,
  REL_TIME                  string,
  CALL_DURATION             string,
```

```
  DURATION                    string,
  CODEC_MODIFY_FLAG           string,
  CODEC_MODIFY_RESULT         string,
  CODEC_NEGOTIATION_FLAG      string,
  CODEC_NEGOTIATION_RESULT    string,
  CODEC_TYPE                  string,
  CALL_TYPE                   string,
  IS_EXT_PLATFORM             string,
  CALL_HOLD                   string,
  CALL_FORWARD                string,
  CALL_WAITING                string,
  CONFRENCE_CALL              string
)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES
("hbase.columns.mapping"=" START_TIME:START_TIME_S,:key,END_TIME:
   END_TIME_S,END_TIME:END_TIME_NS,INDEX:CDR_INDEX,IP:SOURCE_IP,IP:
   DESTINATION_IP,C:CIC,C:OPC,C:DPC,REASON:RELEASE_REASON,NUMBER:
   CALLING_NUMBER,NUMBER:CALLED_NUMBER,NUMBER:ORIGINAL_CALLED_NUMBER,
   NUMBER:TRANSFER_NUMBER,NUMBER:LOCATION_NUMBER,TIME:RESPONSE_TIME,
   TIME:ACM_TIME,TIME:ANM_TIME,TIME:REL_TIME,DURATION:CALL_DURATION,
   DURATION:DURATION,CODEC:CODEC_MODIFY_FLAG,CODEC:CODEC_MODIFY_RESULT
   ,CODEC:CODEC_NEGOTIATION_FLAG,CODEC:CODEC_NEGOTIATION_RESULT,TYPE:
   CODEC_TYPE,TYPE:CALL_TYPE,PLATFORM:IS_EXT_PLATFORM,CALL:CALL_HOLD,
   CALL:CALL_FORWARD,CALL:CALL_WAITING,CALL:CONFRENCE_CALL")
TBLPROPERTIES ("hbase.table.name" = "BICC_CDR");
```

同理这里指定了各个列的对应关系。

2. 插入数据

```
INSERT OVERWRITE TABLE src_tbd_bicc_cdr SELECT a.* FROM bicc a;
```

将数据从表 bicc 中拷贝到表 src_tbd_bicc_cdr 中。

```
hive> INSERT OVERWRITE TABLE src_tbd_bicc_cdr SELECT a.* FROM bicc a;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks is set to 0 since there's no reduce operator
Starting Job = job_201203240810_0008, Tracking URL = http://cloud1:50030/jobdeta
ils.jsp?jobid=job_201203240810_0008
Kill Command = /home/cloud/hadoop/bin/../bin/hadoop job  -Dmapred.job.tracker=cl
oud1:9001 -kill job_201203240810_0008
Hadoop job information for Stage-0: number of mappers: 1; number of reducers: 0
2012-03-24 10:30:46,052 Stage-0 map = 0%,  reduce = 0%
2012-03-24 10:30:49,063 Stage-0 map = 100%,  reduce = 0%
2012-03-24 10:30:52,073 Stage-0 map = 100%,  reduce = 100%
Ended Job = job_201203240810_0008
100 Rows loaded to src_tbd_bicc_cdr
MapReduce Jobs Launched:
Job 0: Map: 1   HDFS Read: 17941 HDFS Write: 0 SUCESS
Total MapReduce CPU Time Spent: 0 msec
OK
Time taken: 13.381 seconds
```

图 2-3　src_tbd_bicc_cdr 表的插入时间

3. 查看插入后的结果：

```
hive> select * from src_tbd_bicc_cdr;
OK
1300532404      53199010        1300532466      843279758       1677724787      135865610       35
136778  1049    NULL    653885  31      13951607180     02557412562                     8625    22
3       2823    0       62732   0       62790   NULL    0       1       1       518     0       22
2       2       2
1300532410      766679302       1300532466      951223166       1677730141      51913994        13
5865610 172     NULL    653873  19      13372033075     13441292651                             85
3       2191    0       56110   0       56184   NULL    0       1       1       518     0       22
2       2       2
1300532412      307871294       1300532466      877261942       1677731328      36250890        51
913994  33      NULL    653890  16      13951634033     13186426580                     8625    16
7       1986    0       54491   0       54569   NULL    0       1       1       518     0       22
2       2       2
1300532413      857059094       1300532466      876262466       1677732472      89466122        69
281034  913     NULL    654117  16      13913928425     13440287355                     8625    59
6       1746    7533    52953   45420   53019   NULL    0       1       1       518     0       22
2       2       2
```

图 2-4　查看 hive 表

再查看 hbase 表：

```
hbase(main):003:0> describe 'BICC_CDR'
DESCRIPTION                                                          ENABLED
 {NAME => 'BICC_CDR', FAMILIES => [{NAME => 'C', BLO                 true
 OMFILTER => 'NONE', REPLICATION_SCOPE => '0', COMPR
 ESSION => 'NONE', VERSIONS => '3', TTL => '21474836
 47', BLOCKSIZE => '65536', IN_MEMORY => 'false', BL
 OCKCACHE => 'true'}, {NAME => 'CALL', BLOOMFILTER =
 > 'NONE', REPLICATION_SCOPE => '0', COMPRESSION =>
 'NONE', VERSIONS => '3', TTL => '2147483647', BLOCK
 SIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE =
 > 'true'}, {NAME => 'CODEC', BLOOMFILTER => 'NONE',
  REPLICATION_SCOPE => '0', COMPRESSION => 'NONE', V
 ERSIONS => '3', TTL => '2147483647', BLOCKSIZE => '
 65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}
 , {NAME => 'DURATION', BLOOMFILTER => 'NONE', REPLI
 CATION_SCOPE => '0', COMPRESSION => 'NONE', VERSION
 S => '3', TTL => '2147483647', BLOCKSIZE => '65536'
```

图 2-5　查看 hbase 表

同理可以对应于上述的大表 yuyin 和 duanxin 建立相应的表 hbase_yuyin 和 hbase_duanxin。以下是它们插入数据所用的时间：

```
Hadoop job information for Stage-0: number of mappers: 1; number of reducers: 0
2012-04-07 18:53:08,925 Stage-0 map = 0%,  reduce = 0%
2012-04-07 18:54:09,033 Stage-0 map = 0%,  reduce = 0%
2012-04-07 18:55:09,155 Stage-0 map = 0%,  reduce = 0%
2012-04-07 18:56:09,264 Stage-0 map = 0%,  reduce = 0%
2012-04-07 18:57:09,392 Stage-0 map = 0%,  reduce = 0%
2012-04-07 18:58:09,500 Stage-0 map = 0%,  reduce = 0%
2012-04-07 18:59:09,623 Stage-0 map = 0%,  reduce = 0%
2012-04-07 18:59:16,637 Stage-0 map = 39%,  reduce = 0%
2012-04-07 19:00:16,746 Stage-0 map = 39%,  reduce = 0%
2012-04-07 19:01:16,855 Stage-0 map = 39%,  reduce = 0%
2012-04-07 19:02:16,965 Stage-0 map = 39%,  reduce = 0%
2012-04-07 19:03:17,077 Stage-0 map = 39%,  reduce = 0%
2012-04-07 19:04:17,184 Stage-0 map = 39%,  reduce = 0%
2012-04-07 19:05:17,288 Stage-0 map = 39%,  reduce = 0%
2012-04-07 19:05:59,361 Stage-0 map = 78%,  reduce = 0%
2012-04-07 19:06:59,467 Stage-0 map = 78%,  reduce = 0%
2012-04-07 19:07:59,580 Stage-0 map = 78%,  reduce = 0%
2012-04-07 19:08:59,683 Stage-0 map = 78%,  reduce = 0%
2012-04-07 19:09:51,780 Stage-0 map = 100%,  reduce = 0%
2012-04-07 19:09:54,787 Stage-0 map = 100%,  reduce = 100%
Ended Job = job_201204071811_0007
3807934 Rows loaded to hbase_yuyin
MapReduce Jobs Launched:
Job 0: Map: 1   HDFS Read: 171862985 HDFS Write: 0 SUCESS
Total MapReduce CPU Time Spent: 0 msec
OK
Time taken: 1012.613 seconds
```

图 2-6　yuyin 表的插入时间

```
2012-04-17 22:38:38,994 Stage-0 map = 75%,  reduce = 0%
2012-04-17 22:39:39,098 Stage-0 map = 75%,  reduce = 0%
2012-04-17 22:40:19,165 Stage-0 map = 88%,  reduce = 0%
2012-04-17 22:41:19,269 Stage-0 map = 88%,  reduce = 0%
2012-04-17 22:42:19,374 Stage-0 map = 88%,  reduce = 0%
2012-04-17 22:43:19,476 Stage-0 map = 88%,  reduce = 0%
2012-04-17 22:44:19,579 Stage-0 map = 88%,  reduce = 0%
2012-04-17 22:45:19,679 Stage-0 map = 88%,  reduce = 0%
2012-04-17 22:46:19,785 Stage-0 map = 88%,  reduce = 0%
2012-04-17 22:47:19,889 Stage-0 map = 88%,  reduce = 0%
2012-04-17 22:48:19,992 Stage-0 map = 88%,  reduce = 0%
2012-04-17 22:49:20,095 Stage-0 map = 88%,  reduce = 0%
2012-04-17 22:50:20,200 Stage-0 map = 88%,  reduce = 0%
2012-04-17 22:51:20,303 Stage-0 map = 88%,  reduce = 0%
2012-04-17 22:51:34,340 Stage-0 map = 100%,  reduce = 0%
2012-04-17 22:51:40,351 Stage-0 map = 100%,  reduce = 100%
Ended Job = job_201204171651_0004
9299126 Rows loaded to hbase_duanxin
MapReduce Jobs Launched:
Job 0: Map: 2   HDFS Read: 439203971 HDFS Write: 0 SUCESS
Total MapReduce CPU Time Spent: 0 msec
OK
Time taken: 3147.16 seconds
```

图 2-7　duanxin 表的插入时间

2.2.3　导入数据到 mongoDB

对于 mongoDB 来说，有其自带的导入工具 mongoimport，这个工具的使用方法可以在官方的文档上找到[2]。由于 mongoDB 面向文档，所以不用事先建立一个表，而是直接可以从文本文件中导入，需要制定相应的域就可以了。这里给出我使用的例子：

```
./mongoimport -d "mydb" -c "bicc" -f "start_time_s,start_time_ns,end_time_t
    ,end_time_ns,cdr_index,source_ip,destination_ip,cic,opc,dpc,
    release_reason,calling_number,called_number,original_called_number,
    transfer_number,location_number,response_time,acm_time,anm_time,rel_time
    ,call_duration,codec_modify_flag,codec_modify_result,
    codec_negotiation_flag,codec_negotiation_result,codec_type,call_type,
```

[2]http://www.mongodb.org/display/DOCS/Import+Export+Tools#ImportExportTools-mongoimport

```
is_ext_platform,call_hold,call_forward,call_waiting,conference_call" --
type csv --file /mnt/hgfs/home/data/bicc_100.csv
```

数据量小时导入很快，当导入数据量很大时就需要消耗一定的时间。如下图所示：从

```
cubika@ubuntu:~/mongodb-linux-i686-2.0.4$ bin/mongoimport -d "mydb" -c "yuyin" -
f "number,mtype,address,calldate,time,ltype,dtype,duration,location_code,base_co
de" --type csv --file /tmp/yuyin.csv
connected to: 127.0.0.1
                280274/164238923        0%
                        6500    2166/second
                720020/164238923        0%
                        16700   2783/second
```

图 2-8　yuyin 表的插入时间

图中可以看出整个导入过程花了 3000 秒左右。

2.2.4　导入数据到 mysql 中

导入时首先建立表，再进行插入。建表语句为：

```
create table duanxin
(
key int,
number string,
mtype int,
address string,
calldate string,
time string,
ltype int,
dtype int,
duration int,
location_code string,
base_code string
)
row format delimited
fields terminated by '|'
stored as textfile;
```

插入语句为：

```
LOAD DATA LOCAL INPATH '/home/cloud/data/ 清单数据/ 数据部
    分/duanxin2.txt' OVERWRITE INTO TABLE duanxin;
```

第三章　查询实验结果

根据上一章提到的研究思路，在导入了数据之后就可以在其之上进行对比性的查询实验。查询时尽量采用一些典型的测试用例，场景分为大小两种数量级的简单和复杂的情形。总共是四种情况，下面先来看小数量级的简单查询情形。

3.1　简单查询

3.1.1　小数量级

这个实验使用了已经建好的表格 bicc, 它只有 100 个记录，简单查询时采用了两个具体的测试用例。其中第一个用例是：

```
select cic from bicc where cic>100 and cic<500;
```

很简单的一个句子，然后我们来看一下这个查询分别在 hive、hbase、mysqL 和 mongoDB 下所得到的实验结果:

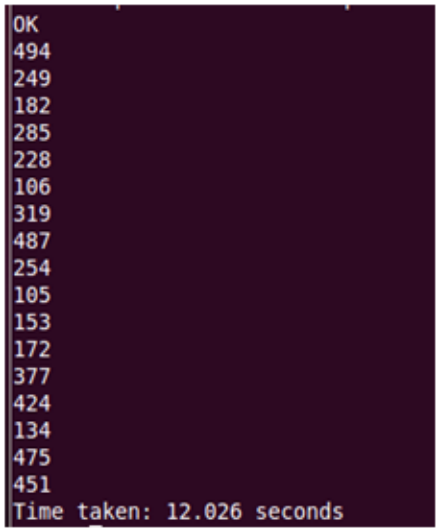


图 3-1　hive 表的查询时间

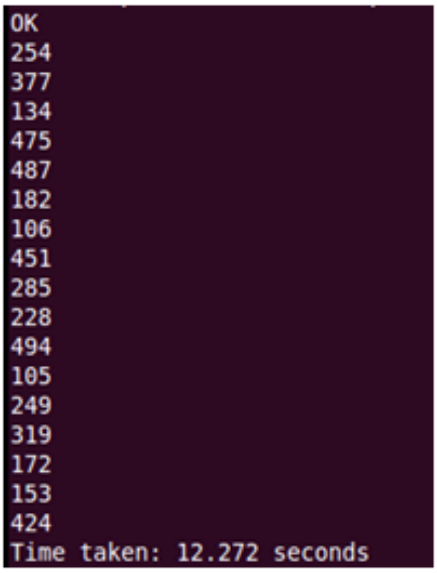


图 3-2　hive/hbase 表的查询时间

```
mysql> select cic from bicc where cic>100 and cic<500;
+------+
| cic  |
+------+
| 494  |
| 249  |
| 182  |
| 285  |
| 228  |
| 106  |
| 319  |
| 487  |
| 254  |
| 105  |
| 153  |
| 172  |
| 377  |
| 424  |
| 134  |
| 475  |
| 451  |
+------+
17 rows in set (0.00 sec)
```

图 3-3　mysql 表的查询时间

```
> show profile

query   mydb.bicc 2ms Sat Apr 07 2012 13:07:17
query:{ "cic" : { "$gt" : 100, "$lt" : 500 } } nscanned:101 nreturned:17 respons
eLength:547 client:127.0.0.1 user:
```

图 3-4　mongoDB 表的查询时间

从这个实验得到的结果可以看出，hive 和 hive/hbase 的结果是一个数量级，均在 12s 左右，而 mysql 和 mongoDB 所耗费的时间在一个数量级，差不多是 3ms 以下。他们之间的差别还是很大的。

第二个用到的测试用例是：

```
select cic from bicc where cic<100 order by cic desc;
```

即从表 bicc 中挑出 cic 列值小于 100 的记录并且递减排序。这个查询最后得到的结果如图 3-5 到图 3-8 所示：

```
Total MapReduce CPU Time Spent: 0 msec
OK
98
86
70
62
60
51
33
32
1
Time taken: 21.265 seconds
```

图 3-5　hive 表的查询时间

```
Total MapReduce CPU Time Spent: 0 msec
OK
98
86
70
62
60
51
33
32
1
Time taken: 20.293 seconds
```

图 3-6　hive/hbase 表的查询时间

```
mysql> select cic from bicc where cic<100 order by cic desc;
+------+
| cic  |
+------+
| 98   |
| 86   |
| 70   |
| 62   |
| 60   |
| 51   |
| 33   |
| 32   |
| 1    |
+------+
9 rows in set (0.12 sec)
```

图 3-7　mysql 表的查询时间

```
> show profile

query   mydb.bicc 2ms Sat Apr 07 2012 13:10:55
query:{
        "query" : {
                "cic" : {
                        "$lt" : 100
                }
        },
        "orderby" : {
                "cic" : -1
        }
} nscanned:101 scanAndOrder nreturned:9 responseLength:299 client:127.0.0.1 user
```

图 3-8　mongoDB 表的查询时间

从第二个测试用例中得到的结果可以看出和由第一个测试用例得出的结果是一致的。

3.1.2　大数量级

在小型数据级上做的实验放到大数据集上又会有不一样的结论。在原来基础上将实验用到的数据量加到很大，具体来说这里用到了已经建好的表 yuyin，它拥有 380 万个记录，以及规模更大的表 duanxin，它拥有 940 万个记录。同样也是采取两个测试用例加以互相验证，其中使用的第一个测试用例为：

```
select number from yuyin where ltype=4 and dtype=4;
```

这个语句很简单，实验的结果可以从图 3-9 到图 3-12 看出：

```
19992
13180
12208
12995
15020
10116
15284
15240
15306
18221
11130
10806
19588
12237
10116
15016
10506
15520
18287
18040
15104
14971
15472
18306
11555
12109
10425
Time taken: 88.36 seconds
```

图 3-9　hive 表的查询时间

```
12536
17175
10525
10355
11998
17982
17930
11767
12001
15016
19327
15928
16713
13837
15715
19539
12338
18112
10202
15752
14615
12617
13372
18602
18458
16043
12842
Time taken: 1199.968 seconds
```

图 3-10　hive/hbase 表的查询时间

```
| 15240  |
| 15306  |
| 18221  |
| 11130  |
| 10806  |
| 19588  |
| 12237  |
| 10116  |
| 15016  |
| 10506  |
| 15520  |
| 18287  |
| 18040  |
| 15104  |
| 14971  |
| 15472  |
| 18306  |
| 11555  |
| 12109  |
| 10425  |
+--------+
979091 rows in set (13.51 sec)
```

图 3-11　mysql 表的查询时间

```
query   mydb.yuyin 134ms Mon Apr 16 2012 13:08:39
query:{ "ltype" : 4, "dtype" : 4 } nscanned:466 nreturned:101 responseLength:224
2 client:127.0.0.1 user:
```

图 3-12　mongoDB 表的查询时间

从上面几个图中可以清晰的看到，hive 查询用了 88s, 而 hive/hbase 查询足足用了 1200s,mysql 用了 13s，mongoDB 由于只查询头 20 条结果，因此时间不足一秒。可以看出 hive 相对来说较之前有所提升，而 hive/hbase 则相对下降，所花费的时间也达到了很难容忍的地步。为了对这个结果加以验证，我们又使用了第二个测试用例：

```
select distinct number from yuyin where duration<60 order by number desc;
```

实验的结果可以从图 3-13 到图 3-16 看出：

```
10034
10032
10031
10030
10029
10028
10027
10025
10024
10023
10022
10020
10019
10017
10016
10015
10014
10013
10012
10010
10009
10005
10004
10003
10002
10001
1#####
Time taken: 56.203 seconds
```

图 3-13　hive 表的查询时间

```
10032
10031
10030
10029
10028
10027
10025
10024
10023
10022
10020
10019
10017
10016
10015
10014
10013
10012
10010
10009
10005
10004
10003
10002
10001
1#####
Time taken: 1171.642 seconds
```

图 3-14　hive/hbase 表的查询时间

```
| 10025  |
| 10024  |
| 10023  |
| 10022  |
| 10020  |
| 10019  |
| 10017  |
| 10016  |
| 10015  |
| 10014  |
| 10013  |
| 10012  |
| 10010  |
| 10009  |
| 10005  |
| 10004  |
| 10003  |
| 10002  |
| 10001  |
| 1##### |
+--------+
8097 rows in set (12.08 sec)
```

图 3-15　mysql 表的查询时间

```
> db.yuyin.find({'duration':{$lt:60}},{number:1}).sort({number:-1});
error: {
        "$err" : "too much data for sort() with no index.  add an index or specify a smaller limit
",
        "code" : 10128
}
```

图 3-16　mongoDB 表的查询时间

从上面结果可以看出，hive 用了 56s，hive/hbase 用了 1171s，mysql 用了 12s，整体来说还是和第一个测试用例得到的结果是一致的。即时间 hive/hbase>hive>mysql。

380 万行的数据查询已经说明了一些问题，如果继续地增加数据量会有什么变化呢？接下来测试一下 940 万行的表 duanxin, 这里采用的测试用例为：

```
select distinct address from duanxin where mtype=0 and ltype=0;
```

实验的结果可以从图 3-17 到图 3-19 看出：

```
hive> select distinct address from duanxin where mtype=0 and ltype=0;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks not specified. Estimated from input data size: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapred.reduce.tasks=<number>
Starting Job = job_201204180844_0001, Tracking URL = http://cloud1:50030/jobdetails.jsp?jobid=job_
201204180844_0001
Kill Command = /home/cloud/hadoop/bin/../bin/hadoop job  -Dmapred.job.tracker=cloud1:9001 -kill jo
b_201204180844_0001
Hadoop job information for Stage-1: number of mappers: 2; number of reducers: 1
2012-04-18 08:49:25,256 Stage-1 map = 0%,  reduce = 0%
2012-04-18 08:49:40,328 Stage-1 map = 32%,  reduce = 0%
2012-04-18 08:49:49,361 Stage-1 map = 64%,  reduce = 0%
2012-04-18 08:49:52,378 Stage-1 map = 75%,  reduce = 0%
2012-04-18 08:49:58,403 Stage-1 map = 88%,  reduce = 0%
2012-04-18 08:50:01,418 Stage-1 map = 100%,  reduce = 17%
2012-04-18 08:50:07,449 Stage-1 map = 100%,  reduce = 100%
Ended Job = job_201204180844_0001
MapReduce Jobs Launched:
Job 0: Map: 2  Reduce: 1   HDFS Read: 439203971 HDFS Write: 9 SUCESS
Total MapReduce CPU Time Spent: 0 msec
OK
0769
769
Time taken: 56.905 seconds
```

图 3-17　hive 表的查询时间

```
2012-04-18 09:24:01,285 Stage-1 map = 58%,  reduce = 19%
2012-04-18 09:25:01,400 Stage-1 map = 58%,  reduce = 19%
2012-04-18 09:26:01,520 Stage-1 map = 58%,  reduce = 19%
2012-04-18 09:27:01,636 Stage-1 map = 58%,  reduce = 19%
2012-04-18 09:27:58,743 Stage-1 map = 67%,  reduce = 19%
2012-04-18 09:28:04,758 Stage-1 map = 67%,  reduce = 22%
2012-04-18 09:28:19,790 Stage-1 map = 75%,  reduce = 22%
2012-04-18 09:28:28,808 Stage-1 map = 75%,  reduce = 25%
2012-04-18 09:29:28,924 Stage-1 map = 75%,  reduce = 25%
2012-04-18 09:30:29,036 Stage-1 map = 75%,  reduce = 25%
2012-04-18 09:31:29,154 Stage-1 map = 75%,  reduce = 25%
2012-04-18 09:32:29,266 Stage-1 map = 75%,  reduce = 25%
2012-04-18 09:33:29,380 Stage-1 map = 75%,  reduce = 25%
2012-04-18 09:34:29,496 Stage-1 map = 75%,  reduce = 25%
2012-04-18 09:34:50,536 Stage-1 map = 83%,  reduce = 25%
2012-04-18 09:34:56,549 Stage-1 map = 92%,  reduce = 25%
2012-04-18 09:34:59,556 Stage-1 map = 92%,  reduce = 28%
2012-04-18 09:35:05,568 Stage-1 map = 92%,  reduce = 31%
2012-04-18 09:36:05,680 Stage-1 map = 92%,  reduce = 31%
2012-04-18 09:37:05,792 Stage-1 map = 92%,  reduce = 31%
2012-04-18 09:38:05,901 Stage-1 map = 92%,  reduce = 31%
2012-04-18 09:39:06,009 Stage-1 map = 92%,  reduce = 31%
2012-04-18 09:40:06,118 Stage-1 map = 92%,  reduce = 31%
2012-04-18 09:41:06,228 Stage-1 map = 92%,  reduce = 31%
2012-04-18 09:41:47,313 Stage-1 map = 100%,  reduce = 31%
2012-04-18 09:41:56,329 Stage-1 map = 100%,  reduce = 100%
Ended Job = job_201204180844_0002
MapReduce Jobs Launched:
Job 0: Map: 12  Reduce: 1   HDFS Read: 0 HDFS Write: 9 SUCESS
Total MapReduce CPU Time Spent: 0 msec
OK
0769
769
Time taken: 2814.768 seconds
```

图 3-18　hive/hbase 表的查询时间

```
mysql> select distinct address from duanxin where mtype=0 and ltype=0;
+---------+
| address |
+---------+
| 769     |
| 0769    |
+---------+
2 rows in set (2 min 32.94 sec)
```

图 3-19　mysql 表的查询时间

从上三个图中可以看出，hive 查询用了 56 秒，hive/hbase 用了 2814 秒，mysql 用了 153 秒。hive 相对于 mysql 来说性能逐渐上升，到这时已经超出 mysql 达到最快，而 hive/hbase 这种方法的性能更加恶化，将近用了 46 分钟的时间，而 hive 花了不到一分钟。可想差别有多大。

3.2　复杂查询

复杂查询有很多种，比如嵌套查询等等。但最为典型的就是两个表之间的连接查询了。因此接下来的所有复杂查询都是指连接查询。而连接查询又分为很多种，如内连接，外连接等等。我们这里只考虑等值连接的情况。

3.2.1　小数量级

由于表 bicc 和 bssap 表的结构相似，我们首先将这两个表进行等值连接。选取 start_time_s 这一项作为比较项。SQL 语句为：

```
select distinct a.start_time_s join bssap b on a.start_time_s=b.
    start_time_s;
```

实验的结果可以从图 3-20 到图 3-22 看出：

```
OK
1300532414
1300532427
1300532430
1300532435
1300532443
1300532451
1300532453
1300532454
1300532455
1300532456
1300532457
1300532458
1300532459
1300532460
1300532461
1300532462
1300532463
Time taken: 45.283 seconds
```

图 3-20　hive 表的查询时间

```
1300532414
1300532427
1300532430
1300532435
1300532443
1300532451
1300532453
1300532454
1300532455
1300532456
1300532457
1300532458
1300532459
1300532460
1300532461
1300532462
1300532463
Time taken: 42.418 seconds
```

图 3-21　hive/hbase 表的查询时间

```
mysql> select distinct a.start_time_s from bicc a join bssap b on a.start_time_s=b.start_time_s;
+--------------+
| start_time_s |
+--------------+
| 1300532460   |
| 1300532430   |
| 1300532459   |
| 1300532443   |
| 1300532455   |
| 1300532456   |
| 1300532458   |
| 1300532462   |
| 1300532451   |
| 1300532463   |
| 1300532461   |
| 1300532435   |
| 1300532457   |
| 1300532453   |
| 1300532454   |
| 1300532414   |
| 1300532427   |
+--------------+
17 rows in set (0.05 sec)
```

图 3-22　mysql 表的查询时间

由于这两个表都非常小，join 时花费的时间还是很少的。但仍然可以看出 mysql 远远快于 hive 和 hive/hbase。

3.2.2　大数量级

将已经建好的表 yuyin 和表 duanxin 进行 join 等值连接：

```
select count(distinct a.address) from yuyin a join duanxin b on a.number=b.
    number;
```

由于表的规模太大，所以采用了让程序在服务器后台运行的方法。为了准确计算时间，我编写了 shell 脚本调用各个程序并计算调用程序前后的时间差，转换后的结果为秒。图 3-23 和 3-24 显示了 hive 和 hive/hbase 的脚本。

```
#/bin/sh -f
date_start=`date "+%s"`
bin/hive -e "select count(distinct a.number) from yuyin a join duanxin b on a.number = b.number"
date_end=`date "+%s"`
timespend=`expr "$date_end" - "$date_start"`
echo "this app start at $date_start,and end at $date_end"
echo "程序运行了：$timespend 秒"
```

图 3-23　hive 执行脚本

```
#/bin/sh -f
date_start=`date "+%s"`
bin/hive -e "select count(distinct a.number) from hbase_yuyin a join hbase_duanx
in b on a.number = b.number"
date_end=`date "+%s"`
timespend=`expr "$date_end" - "$date_start"`
echo "this app start at $date_start,and end at $date_end"
echo "程序运行了：$timespend 秒"
```

图 3-24　hive/hbase 执行脚本

后台执行脚本时输入：

```
nohup ./h1.sh > h1out.txt 2>&1 &;
```

这样执行的结果就重定向到 h1out.txt 文本文件中。

mysql 执行时也采取同样的办法。但是 mysql 第一次运行时出现“MySQL client ran out of memory”的错误。经过查阅资料，发现 mysql 分别使用 mysql_store_result 和 mysql_use_result 这两种方法存储结果集。mysql_store_result 方法是把结果集全部存储在内存中，而 mysql_use_result 不把结果集存在客户端内存中，而是按需去 mysql 服务器取。由于这个连接查询结果集太大，如果用 mysql_store_result 的方法是放不下内存的，所以需要改用第二个方案。在执行时加入 –quick 选项。除此之外，由于结果太大，磁盘空间很可能不足，可以采用 mysql 提供的格式化结果的选项，使用 grep 命令只将带有时间的一行输出。

本次使用了 count 命令使得程序的输出没有受到磁盘空间的困扰，完整的 mysql 执行脚本为：

```
#/bin/sh -f
date_start=`date "+%s"`
mysql -uroot -p123456 -D mydb -e "select count(distinct a.number) from yuyin a join duanxin b on a
.number = b.number"
date_end=`date "+%s"`
timespend=`expr "$date_end" - "$date_start"`
echo "this app start at $date_start,and end at $date_end"
echo "程序运行了：$timespend 秒"
```

图 3-25　mysql 执行脚本

最后，实验的结果可以从图 3-26 到图 3-28 看出：

```
Starting Job = job_201205051406_0004, Tracking URL = http://cloud1:50030/jobdetails.jsp?jobid=job_
201205051406_0004
Kill Command = /home/cloud/hadoop/bin/../bin/hadoop job  -Dmapred.job.tracker=cloud1:9001 -kill jo
b_201205051406_0004
Hadoop job information for Stage-2: number of mappers: 1; number of reducers: 1
2012-05-05 14:51:05,732 Stage-2 map = 0%,  reduce = 0%
2012-05-05 14:51:08,738 Stage-2 map = 100%,  reduce = 0%
2012-05-05 14:51:17,754 Stage-2 map = 100%,  reduce = 100%
Ended Job = job_201205051406_0004
MapReduce Jobs Launched:
Job 0: Map: 3  Reduce: 1   HDFS Read: 640419324 HDFS Write: 196153 SUCESS
Job 1: Map: 1  Reduce: 1   HDFS Read: 196341 HDFS Write: 5 SUCESS
Total MapReduce CPU Time Spent: 0 msec
OK
8089
Time taken: 1807.276 seconds
this app start at 1336198872,and end at 1336200680
程序运行了：1808 秒
```

图 3-26　hive 表的查询时间

```
Starting Job = job_201205051406_0006, Tracking URL = http://cloud1:50030/jobdeta
ils.jsp?jobid=job_201205051406_0006
Kill Command = /home/cloud/hadoop/bin/../bin/hadoop job  -Dmapred.job.tracker=cl
oud1:9001 -kill job_201205051406_0006
Hadoop job information for Stage-2: number of mappers: 1; number of reducers: 1
2012-05-06 09:27:37,555 Stage-2 map = 0%,  reduce = 0%
2012-05-06 09:27:40,562 Stage-2 map = 100%,  reduce = 0%
2012-05-06 09:27:49,577 Stage-2 map = 100%,  reduce = 100%
Ended Job = job_201205051406_0006
MapReduce Jobs Launched:
Job 0: Map: 16  Reduce: 1   HDFS Read: 0 HDFS Write: 196153 SUCESS
Job 1: Map: 1  Reduce: 1   HDFS Read: 196341 HDFS Write: 5 SUCESS
Total MapReduce CPU Time Spent: 0 msec
OK
8089
Time taken: 5487.677 seconds
this app start at 1336262183,and end at 1336267673
程序运行了：5490 秒
```

图 3-27　hive/hbase 表的查询时间

```
nohup: 忽略输入
count(distinct a.number)
8089
this app start at 1336207369,and end at 1336219313
程序运行了：11944 秒
~
```

图 3-28　mysql 表的查询时间

可以看出 hive 用了 1808 秒，hive/hbase 用了 5490 秒，mysql 用了 11944 秒。mysql 执行的时间明显高于其他。

3.3　总结

	hive	hive/hbase	mysql
简单查询（100 行）	12s-20s	12s-20s	0ms-100ms
简单查询（380 万行）	88s	1200s	14s
简单查询（940 万行）	57s	2814s	153s
复杂查询（小数量级)	45s	42s	5ms
复杂查询（大数量级）	1808s	5490s	11944s

表 3-1　实验数据汇总

第四章　查询优化

在实际的操作中，不同的查询需要采取不同的实现方法。有时候查询语句往往一个小的改动都会造成完全不一样的时间消耗，这一点在大型数据上的表现尤其的明显。所以，在实际的执行过程中要特别注重查询的效率问题，结合数据库理论和具体的环境进行不同的优化工作。优化的方法有很多种，根据我的查找和思考，以及[6],[7],[8],[9]的介绍，这里仅仅介绍我个人认为比较重要的几个方面。

4.1　使用索引

我们知道，索引是数据库中十分重要的一个概念，如果使用的合理适当，能够极大幅度的提高效率。从某种意义上来说，索引可以理解为一种特殊的目录。索引中保存着表中存储的索引列，并且记录了索引列在数据库表中的物理存储位置，实现了表中数据的逻辑索引。在没有索引时，查询需要进行全表的扫描，将每一条的记录取出并和查询条件一一对比，这无疑会消耗大量的时间并且造成许多磁盘 IO。有了索引之后，就可以直接在索引中找到符合条件的索引值，将相应的指针返回就能够找到需要的记录。因此，有了索引可以避免全表扫描，也就可以大大减少时间。这和通过目录查找相应的内容的道理是一样的，要比一页一页的翻肯定来得快。

需要注意的是，索引也是一个单独的物理结构，建立索引需要花费空间和时间的代价，它也是要占用物理空间的。更加重要的是，一旦表中的记录发生了变化如增加删除和修改等等，索引本身也要相应的调整，因此要适当考虑创建和维护索引的代价，想好究竟哪些字段需要建立索引，不能一味建立索引。

事实上，索引的建立是有一定的原则的。如对于查询中很少涉及到的列或者重复值比较多的列不需要建立索引；对于按范围查询的列最好建立索引；表中若有主键或者外键一定要建立索引；而对于一些特殊的数据类型，如文本字段 TXT，图像类型字段 IMAGE 等等，由于长度不定，很难建立索引。

一般数据库中的索引分为唯一索引、主键索引和聚集索引。唯一索引就是不允许任何两行有相同索引值的索引；主键索引是为主键所准备，它唯一表示了这一行；聚集索引是使得表的索引顺序和物理顺序一样，一个表只能包含一个聚集索引。

根据索引的特点一般来说，下面的几种情况下建立索引最为有效：

- 在经常进行连接的字段上建立索引。
- 在频繁进行 groupby 和 orderby，即排序或者分组操作的列上建立索引。
- 在条件表达式中经常用到的不同值较多的列上建立索引。
- 如果待排序的列有多个，可以在这些列上建立复合索引。

接下来我们通过一个例子来验证索引的作用：

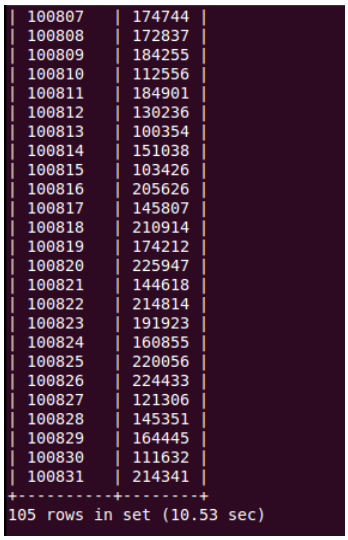
```
// 查询语句
select calldate,time from duanxin where calldate>100600 and time>110000
   group by calldate;

// 建立索引
create index date_time on duanxin(calldate(6),time(6));

// 使用索引
```

```
select calldate,time from duanxin force index(date_time) where calldate
   >100600 and time>110000 group by calldate;
```

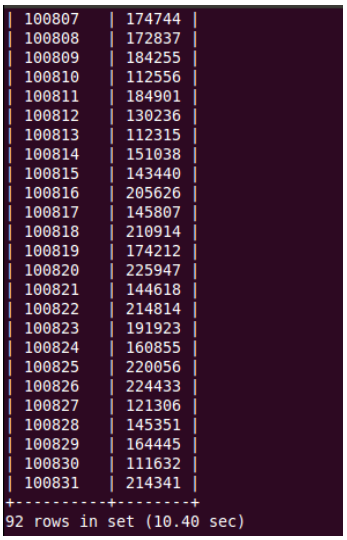
下面是建立索引前后的对比：



100807	174744
100808	172837
100809	184255
100810	112556
100811	184901
100812	130236
100813	100354
100814	151038
100815	103426
100816	205626
100817	145807
100818	210914
100819	174212
100820	225947
100821	144618
100822	214814
100823	191923
100824	160855
100825	220056
100826	224433
100827	121306
100828	145351
100829	164445
100830	111632
100831	214341

105 rows in set (10.53 sec)

图 4-1　建立索引前



100807	174744
100808	172837
100809	184255
100810	112556
100811	184901
100812	130236
100813	112315
100814	151038
100815	143440
100816	205626
100817	145807
100818	210914
100819	174212
100820	225947
100821	144618
100822	214814
100823	191923
100824	160855
100825	220056
100826	224433
100827	121306
100828	145351
100829	164445
100830	111632
100831	214341

92 rows in set (10.40 sec)

图 4-2　建立索引后

4.2　参数优化

在实际应用中，设定参数可以调优查询代码的执行效率。这里仅仅讨论 hive 的参数优化问题。

4.2.1　如何设定参数

首先说明的是，对于一般的参数，大概有三种设定方式：

- 配置文件
- 命令行参数
- 参数声明

其中配置文件就是 conf 文件夹下的文件，可以是默认的，也可以是用户自己定义的配置文件。配置文件的设定对本机启动的所有 Hive 进程都有效。而命令行参数就是在启动 hive 时，在命令行添加 -hiveconf param=value 来设定。参数声明就是在 HQL 中使用 SET 关键字设定，如 set mapred.reduce.tasks=100. 后两种设定的作用域是 Session 级的。

4.2.2　列裁剪（Column Pruning）

在读数据的时候，只读取查询中需要的列，而忽略其他列。例如表 T 包含 5 个列 (a,b,c,d,e)，当实现查询

```
SELECT a,b FROM T WHERE e < 10;
```

这样列 c，d 将会被忽略，只会读取 a, b, e 列。这个选项默认为真：hive.optimize.cp = true

4.2.3　分区裁剪（Partition Pruning）

在查询时减少不必要的分区。例如：

```
SELECT * FROM (SELECT c1, COUNT(1)
  FROM T GROUP BY c1) subq
  WHERE subq.prtn = 100;

SELECT * FROM T1 JOIN
  (SELECT * FROM T2) subq ON (T1.c1=subq.c2)
  WHERE subq.prtn = 100;
```

会在子查询中就考虑 subq.prtn = 100 条件，从而减少读入的分区数目。此选项默认为真：hive.optimize.pruner=true。

4.2.4 Join 和 MapJoin

首先应该将条目少的表/子查询放在 Join 操作符的左边。因为 join 操作符左边的表的内容会被加载进内存。另外，在多个表进行 join 时，如果 join 的 key 值是相同的，那么将会合并成一个 MapReduce 任务，如果不同，则任务数目和 join 操作的数目相同。

join 的方法有 reduce 端的 join 和 map 端的 join 之分，其中 reduce 端的 join 的工作就是把两个表都生成 key-value 对，在 reduce 端进行笛卡尔积；mapjoin 的原理就是把小表全部读入内存中，在 map 阶段直接拿另外一个表的数据和内存中表数据做匹配。换句话说，只将大表生成 key-value 对，由于小表非常的小，足以在内存中放下并且可以进行全表的扫描，因此在 map 端就可以将小表中的值全部取出并匹配，避免了笛卡尔乘积。[10].

下面是一个使用 mapjoin 的例子：

```
SELECT /*+ MAPJOIN(x) */ x.key, x.value, y.valueFROM src1 x LEFT OUTER JOIN
    src y ON (x.key = y.key);
```

再次强调 mapjoin 仅仅适用于一个小表和一个大表 join 的场景。因此这里的表 x 一定是小表。如果太大会出现内存不足的错误。

4.2.5 groupby

hive.map.aggr = true 是否在 Map 端进行聚合，默认为 True hive.groupby.mapaggr.checkinterval = 100000 在 Map 端进行聚合操作的条目数目 hive.groupby.skewindata = false 有数据倾斜的时候进行负载均衡

4.2.6 合并小文件

文件数目过多，会给 HDFS 带来压力，并且会影响处理效率，可以通过合并 Map 和 Reduce 的结果文件来消除这样的影响：

- hive.merge.mapfiles = true 是否和并 Map 输出文件，默认为 True
- hive.merge.mapredfiles = false 是否合并 Reduce 输出文件，默认为 False
- hive.merge.size.per.task = 256*1000*1000 合并文件的大小

4.3 解决数据倾斜

数据倾斜的原因是在 map 端处理数据量的差异太大，进而导致数据无法均匀的分配到各个 reduce 中。数据倾斜有很多种情形，比如说 join 时的 key 值分布不均匀，或者 join 时存在很多的空值。因为 Hive 中，主键为 null 值的项会被当做相同的 Key 而分配进同一个计算 Map。另外还有 groupby 或者 count distinct 时某些值太多导致处理这个

值的 reduce 非常的耗时。解决的方法有很多种，比如对于 join 时存在很多空值的情况，在 sql 语句中设置值不为空的不参与关联，或者把空值的 key 变成一个字符串加上随机数，把倾斜的数据分到不同的 reduce 上，由于 null 值关联不上，处理后并不影响最终结果。对于 count distinct 情况，可以利用 group by 替代它达到效果。比如说这样改：

```
//count(distinct)sql 语句
select address,count(distinct number)
from duanxin
group by address;替换的

//group by 语句
select address,count(1)
from (
select address
from duanxin
group by address,number
) t
group by address;
```

下面两个图是两个方法得出的结果。由于这里第一数据量不是特别大，第二数据

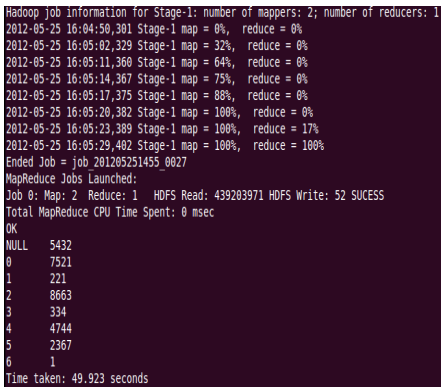


图 4-3　使用 cout(distinct)

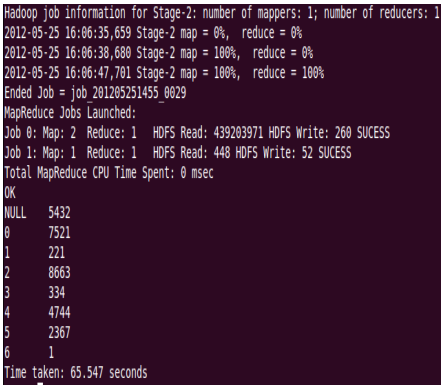


图 4-4　使用 group by 代替之

倾斜不是很严重，第三替换后的增加了子查询的操作所以才会出现前者时间比后者多的情况。

4.4　对 join 查询算法的研究

参考文献[11] 比较详细的介绍了各种 join 的算法。标准的 join 算法可以通过 MapReduce 的一个 Job 实现。在 map 阶段通过提取需要 join 的 key，生成一个标签 tag, 再加上 value 形成一个（key,value) 对。这个标签标志了这条记录来源于哪一个表。接下来将他们进行排序与合并。在 reduce 阶段，对于每一个 key，根据标签来把每一个 value 值划分为左右两个缓冲区，该记录来源于两个表中的哪个表就放到哪个缓冲区里。然后进行笛卡尔积就可以得到结果。伪代码可以表示为：

```
(
MapK:null,V:a record from a split of either R or L)
  join_key ← extract the join column from V
  tagged_record ← add a tag of either R or L to V (
  emitjoin_key,tagged_record)

Reduce( K: a join key, LISTV: r ecords from R and L with join key  K)
```

```
create buffers BR and BL for R and L, respectively
for each record t in LISTVdo
append t to one of the buffers according to its tag
for each pair of records (r, l) in ×BRBL do
emit(null , newrecord(r,l))
```

这个算法的实现在 hadoop 自带的例子 datajoin 里面有。代码如下：

```
//SampleDataJoinMapper.java
public class SampleDataJoinMapper extends DataJoinMapperBase {


  protected Text generateInputTag(String inputFile) {
    // tag the row with input file name (data source)
    return new Text(inputFile);
  }

  protected Text generateGroupKey(TaggedMapOutput aRecord) {
    // first column in the input tab separated files becomes the key (to
       perform the JOIN)
    String line = ((Text) aRecord.getData()).toString();
    String groupKey = "";
    String[] tokens = line.split("\\t", 2);
    groupKey = tokens[0];
    return new Text(groupKey);
  }

  protected TaggedMapOutput generateTaggedMapOutput(Object value) {
    TaggedMapOutput retv = new SampleTaggedMapOutput((Text) value);
    retv.setTag(new Text(this.inputTag));
    return retv;
  }
}

//SampleDataJoinReducer.java
public class SampleDataJoinReducer extends DataJoinReducerBase {
  protected TaggedMapOutput combine(Object[] tags, Object[] values) {
    // eliminate rows which didnot match in one of the two tables (for
       INNER JOIN)
    if (tags.length < 2)
       return null;
    String joinedStr = "";
    for (int i=0; i<tags.length; i++) {
      if (i > 0)
         joinedStr += "\t";
      // strip first column as it is the key on which we joined
      String line = ((Text) (((TaggedMapOutput) values[i]).getData())).
         toString();
      String[] tokens = line.split("\\t", 2);
      joinedStr += tokens[1];
    }
    TaggedMapOutput retv = new SampleTaggedMapOutput(new Text(joinedStr));
    retv.setTag((Text) tags[0]);
    return retv;
  }
}

//SampleDataOutput.java
public class SampleTaggedMapOutput extends TaggedMapOutput {
```

```
  private Text data;

  public SampleTaggedMapOutput() {
    this.data = new Text("");
  }

  public SampleTaggedMapOutput(Text data) {
    this.data = data;
  }

  public Writable getData() {
    return data;
  }

  public void write(DataOutput out) throws IOException {
    this.tag.write(out);
    this.data.write(out);
  }

  public void readFields(DataInput in) throws IOException {
    this.tag.readFields(in);
    this.data.readFields(in);
  }
}
```

这个算法的问题在于对两个表都需要进行缓冲，数据量很大时表不一定能完全放到内存缓冲区里。针对这个问题有许多种改进方法。第一个改进就是将标签放到 value 里，形成 value_tag 的方式，这样便于对两个表分别进行排序。第二个改进就是无论在 map 阶段还是 reduce 阶段都对 join_key 生成 hash 值，这样便于查找。最后的改进方法包括只将相对较小的表进行缓冲，然后再生成 join 的结果。

```
Map(K: null , V: a record from a split of either R or L)
joinkey ← extract the join column from V
tagged record ← add a tag of either R or L to V
compositekey ← (joinkey, tag)
emit(compositekey, taggedrecord)
Partition(K: input key)
hashcode ← hashfunc(K.joinkey)
return hashcode mod # reducers

Reduce( K: a composite key with the join key and the tag
LISTV: records for  K, first from R, then L)
create a buffer BR for R
for each R record r in  LISTVdo
store r in BR
for each L record l in  LISTVdo
for each record r in BR do
emit(null , newrecord(r, l))
```

还有一个改进可以是首先进行预处理操作，针对 join_key 先进行分块，这样在以后的 join 操作之前都避免了相同操作，直接的进行 join 就行了。

第五章　HBase 客户端编程

我们知道，HBase 有很多访问接口，其中，HBase Shell 是 HBase 的命令行工具，它最简单，适合 HBase 管理使用；Thrift Gateway，它利用 Thrift 序列化技术，支持 C++，PHP，Python 等多种语言，适合其他异构系统在线访问 HBase 表数据；REST Gateway，支持 REST 风格的 Http API 访问 HBase, 解除了语言限制；还有就是我们这一章用到的 Native Java API 访问方式，他是最常规和高效的访问方式，适合 Hadoop MapReduce Job 并行批处理 HBase 表数据。用户可以通过编写 Java 程序来对 HBase 中存在的表进行新建删除、对记录进行插入删除以及查询等等。编程时用到的 api 可以从官方网站[1]上查到非常详细的介绍。下面介绍几个客户端编程用到的重要概念：

5.1　客户端编程的重要概念

5.1.1　HBaseConfiguration

HBaseConfiguration 是每一个 hbase client 都会使用到的对象，它代表的是 HBase 配置信息。默认的构造方式会尝试从 hbase-default.xml 和 hbase-site.xml 中读取配置。如果 classpath 没有这两个文件，就需要你自己设置配置。

```
Configuration HBASE_CONFIG = new Configuration();
HBASE_CONFIG.set“(hbase.zookeeper.”quorum, “”zkServer);
HBASE_CONFIG.set“(hbase.zookeeper.property.”clientPort, “2181);
HBaseConfiguration cfg = new HBaseConfiguration(HBASE_CONFIG);
```

5.1.2　HBaseAdmin

HBaseAdmin 负责表的 META 信息处理。创建和删除表都是通过 HBaseAdmin 来操作的。其中创建表时使用 HBaseAdmin 提供的 createTable 这个方法，并可以通过 addFamily 方法增加列族。删除表之前要 disableTable，然后再进行 deleteTable 操作。下面代码是创建表和删除表的示例：

```
//创建表
HBaseAdmin hAdmin = new HBaseAdmin(hbaseConfig);
HTableDescriptor t = new HTableDescriptor(tableName);
t.addFamily(new HColumnDescriptor“(f1));
t.addFamily(new HColumnDescriptor“(f2));
t.addFamily(new HColumnDescriptor“(f3));
t.addFamily(new HColumnDescriptor“(f4));
hAdmin.createTable(t);

//删除表
HBaseAdmin hAdmin = new HBaseAdmin(hbaseConfig);
if (hAdmin.tableExists(tableName)) {
        hAdmin.disableTable(tableName);
        hAdmin.deleteTable(tableName);
}
```

[1]http://hbase.apache.org/apidocs/index.html

5.1.3 Put 和 Get/Scan

Put 方法用来向 HTable 中插入数据，可以通过以下两种方法插入单个数据或者批量插入：

```
public void put(final Put put) throws IOException
public void put(final List<Put> puts) throws IOException
```

get 方法可以实现通过 rowkey 在 table 中查询某一行的数据，getScanner 方法可以通过指定一段 rowkey 范围来查询。

```
public Result get(final Get get)
public ResultScanner getScanner(final Scan scan)
```

get/scan 可以通过 addFamily/addColumn 方法指定 family 或者 column；通过 setFilter 来过滤信息；Scan 可以通过 setStartRow 和 setStopRow 来指定开始和结束的行。结果是 Result 对象，可以通过 result 的 getRow，getValue 等方法获取详细信息。下面是两个例子：

```
//通过 row 获取 data
       public static void getbyrow(HTable table, String row) throws
           IOException {
                  Get get=new Get(row.getBytes());
                  Result r = table.get(get);
                  print(r);
       }
 }

//返回所有data
public static void getAllData(HTable table) throws IOException {
          Scan s = new Scan();
          ResultScanner rs = table.getScanner(s);
           for (Result r : rs) {
                   print(r);
           }
}
```

5.1.4 Filters

Hbase 中的 Filter 是用来进行过滤查询信息的，HBase 本身提供了多种多样的 Filter。比较类型的 Filter 包括 RowFilter，它可以根据 rowkey 来进行过滤，FamilyFilter 可以根据列族的比较来过滤，类似的还有 QualifierFilter 和 ValueFilter。专用的 Filter 包括 SingleColumnValueFilter，它可以根据值来决定是否返回该行，PrefixFilter 可以根据行首信息来过滤，PageFilter 指定一页中可以返回多少行，KeyOnlyFilter 只考虑 key 值而忽略 value 值。此外还有许许多多的 Filter。一次可以使用多个 Filter，可以把不同的 Filter 放到一个 FilterList 里面。

详细信息可以查看[12] 这一本书中更加详细的介绍。

5.2 应用实例

对应前面的条件查询实验，这里我通过使用 SingleColumnValueFilter 来实现 select * from yuyin where ltype=4 and dtype=4 的查询。

5.2.1 代码

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.KeyValue;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.filter.FilterList;
import org.apache.hadoop.hbase.filter.SingleColumnValueFilter;
import org.apache.hadoop.hbase.filter.CompareFilter.CompareOp;
import org.apache.hadoop.hbase.util.Bytes;

import org.apache.hadoop.hbase.HConstants;

public class ValueFilter {

    private static Configuration conf = null;

    /**
     * 初始化配置
     */
    static {
        conf = HBaseConfiguration.create();

//设置timeout 为120s
conf.setLong(HConstants.HBASE_REGIONSERVER_LEASE_PERIOD_KEY, 120000);
    }

 public static void selectByFilter(String tablename,List<String> arr)
     throws IOException{
        HTable table=new HTable(conf,tablename);
        FilterList filterList = new FilterList();
        Scan s1 = new Scan();
        for(String v:arr){ // 各个条件之间是“与”的关系
            String [] s=v.split(",");
            filterList.addFilter(new SingleColumnValueFilter(Bytes.toBytes(
                s[0]),Bytes.toBytes(s[1]),
                        CompareOp.EQUAL,Bytes.toBytes(s[2]) )
            );
            // 添加下面这一行后，则只返回指定的，同一行中的其他不返回cellcell
         s1.addColumn(Bytes.toBytes(s[0]), Bytes.toBytes(s[1]));
        }
        s1.setFilter(filterList);
        ResultScanner ResultScannerFilterList = table.getScanner(s1);
        for(Result rr=ResultScannerFilterList.next();rr!=null;rr=
            ResultScannerFilterList.next()){
            for(KeyValue kv:rr.list()){
                System.out.println("row : "+new String(kv.getRow()));
                System.out.println("column : "+new String(kv.getFamily()));
                System.out.println("value : "+new String(kv.getValue()));
            }
        }
    }
```

```
    public static void  main (String [] agrs) throws IOException {
        long start= System.currentTimeMillis();
        List<String> arr=new ArrayList<String>();
        // select * from yuyin where ltype=4 and dtype=4;
        arr.add("type,ltype,4");
         arr.add “(type,dtype” ,4);
        ValueFilter.selectByFilter("yuyin",arr);
        long end= System.currentTimeMillis();
        System.out.println("time spend: "+(end-start)/1000+"s");
}
}
```

首先将查询条件放到一个 List 里面，将 List 中的字符串通过逗号分开生成字符串数组 s,s[0] 是 hbase 中的 column family，s[1] 是 column family 下对应的 column，s[2] 是相应的 value。通过 scan 对象调用 setFilter 方法进行过滤，通过 table.getScanner 方法得到类型为 ResultScanner 的结果。逐个遍历 ResultScanner 的内容就可以打印出相应的信息。

5.2.2 编译并打包

```
// 编译
javac -cp $HBaseClassPath:$JAVA_HOME/lib/*.jar ValueFilter.java

// 打包
jar -cvf ValueFilter.jar ValueFilter.class
```

编译时指定各种 jar 包，包括 hadoop core 的包，hbase 版本的包，hbase lib 文件夹下的所有 jar 包等等。通过指定这些 jar 包的路径作为 classpath，就能成功的生成 class 文件，即 java 的字节码。编译之后的打包工作是为了 hadoop 调用。

5.2.3 执行

```
~/hadoop/bin/hadoop jar ValueFilter.jar ValueFilter
```

执行的结果为：

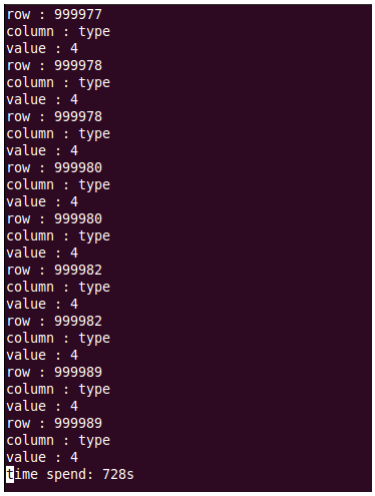


图 5-1　mongoDB 表的查询时间

可以看出，经过了编译、打包、执行等一系列工作，最终的结果是能够准确的返回符合条件的记录集。

参考文献

[1] T. White, *Hadoop:The Definitive Guide*. YAHOO! press, June ,2009.

[2] M. Ian Thomas Varley, "No relation: The mixed blessings of non-relational databases," The University of Texas, Austin, Tech. Rep. MSU-CSE-99-39, August 2009.

[3] N. J. ASHISH T, JOYDEEP S, "Hive-a petabyte scale data warehouse using hadoop," *IEEE 26th International*, vol. 9, no. 996-1005, p. , 2010.

[4] L. X.-y. LIU Yong-zeng, ZHANG Xiao-jing, "基于 hadoop / hive 的 web 日志分析系统的设计," *Journal of Guangxi University*（*Natural Science Edition)*, vol. 36, no. A01, pp. 314–317, OCT 2011.

[5] M. D. Kristina Chodorow, *MongoDB: The Definitive Guide*. O'Reilly Media, April, 2011.

[6] 张正本 and 蔡鹏飞, "海量数据查询优化," 信息与电脑, 2010.

[7] 毛杰 and 佘名高, "海量数据库查询优化研究," 软件导刊 *SOFT WARE GUIDE*, vol. 9, 2010.

[8] 叶文宸, "基于 hive 的性能优化方法的研究与实践," 南京大学, OCT 2011.

[9] . 谭磊 and 王占宏, "基于优化器的提高海量数据查询效率方法研究," 计算机应用与软件, vol. 01, 2012.

[10] A. K. H. T. Dawei Jiang and G. Chen, "Map-join-reduce: Towards scalable and efficient data analysis on large clusters," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 23, no. 9, pp. 1299–1311, SEP 2011.

[11] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in mapreduce," in *Proceedings of the 2010 international conference on Management of data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 975–986. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807273

[12] L. George, *HBase:The Definitive Guide*. O'Reilly Media, September 21, 2011.

致　　谢

本次毕设的完成我得到了很多人的帮助。首先要感谢的是我的指导老师侯宾，从最初的开题到中间的每一周的开展我都得到了侯宾老师非常耐心的指导，不仅使我对自己的毕设有了更深入的了解，而且对于未来工作如何开展、对于整体情况的把握都有非常重要的帮助。另外还要感谢卓海艺学长对我的细心指导和帮助，正是他和我共同探讨了毕设的设计，并且在我遇到问题时给了我正确的纠正，还有一直以来对我的支持。此外还要感谢我的室友闭美春指导我论文格式的排版工作。

本次毕设所做的工作相对来说比较基础，由于个人能力和时间等问题，特别是在优化这方面还有不少可以改进的地方，希望在未来能够加以改善。

外 文 译 文

Distributed Semantic Web Data Management in HBase and MySQL Cluster

By Craig Franke, Samuel Morin, Artem Chebotko, John Abraham, and Pearl Brazier

在 HBase 和 MySQL 集群中的分布式语义 WEB 的数据管理

摘要

在互联网上的各种各样的计算型资源和数据资源正通过机器解释的语义描述方式得到增强，以便于更好的搜索、发现和集成。这个由相互关联的元数据构成的语义 Web，它的容量能潜在地增加着整个互联网的规模。语义 Web 数据的高效管理，通过 W3C 的资源描述框架进行表述，对于支持新型的数据密集型语义的应用来说是非常关键的。在本次工作中，我们研究并比较了两种基于新兴的云计算技术和传统的关系型数据集群技术的分布式 RDF 数据管理方法。特别的，我们为 HBase 和 MySQL 集群设计了分布式 RDF 数据存储和查询方案，并且在 the Third Provenance Challenge 和 Lehigh 大学的机器集群上通过使用基准数据集和查询对这些方法进行了实证比较。我们的研究结果揭示了有趣的查询评估模式，这表明了我们的算法是拥有前途的，而且暗示了对于可扩展的语义 Web 数据管理来说云计算有很大的潜力。

综述

万维网联盟 (W3C) 推荐并标准化了一系列的规则则、语言、框架和连接各种各样的元数据到下一代互联网的最佳实践，那就是语义 Web。W3C 的元数据获取语言包括资源描述框架（RDF），RDFa(属性),RDFs(模式)，Web 本体语言 (OWL)。政府、学术界和产业界积极的拥抱这些技术来捕获和共享语义 Web 的元数据。举几个例子，oeGOV 正在为电子市场制定和发布 OWL 本体，美国人口普查数据正以 RDF 的形式发布，生物信息学家以 RDF 的方式维持通用蛋白质资源，地球科学家出版全球范围的地里的 RDF 数据库 GeoNames，美国最大的电子产品零售商百思买以 RDF 的方式发布它所有的产品目录，美国最大的社交网络提供商 Facebook 使用 RDFa 在它的网页中嵌入元数据。服务计算社区通过使用了例如 OWL-S、WSDL-S、SWSO 等词汇的语义注释，增强了现有的语义 Web 服务。

RDF 数据模型是一个直接的，标识的图形，他也可以被序列化为一组三元关系。本论文中的例子包括 10 个描述了使用 LUBM 词汇的三元组，如图 1 所示。每个三元组由一个主体，谓词和对象组成，并且定义了一个主体和对象之间的关系。图中 <> 和“ ”分别表示资源标识符和一些个数据类型。例如，前三个三元组的资源标识符 C 是一个学生，拥有名字 Craig 并且是 IEEE 的成员。这个示例数据集可以使用 SPARQL 语言来查询，它是一个 RDF 的标准查询语言。SPARQL 使用三元模型和图形模型来匹配 RDF 数据。例如从包含?X< type ><UndergraduateStudent> 模式的 LUBM 中的 Q14 查询返回所有本科生标志并赋给 X 变量。关于 SPARQL 特性和语义更多的细节可以在 W3C

的 SPARQL 规范中发现。

随着语义 Web 的快速发展和作为元数据的主要语言的 RDF 的广泛应用，RDF 数据有效的管理将会对于支持新型的在不同领域的语义应用来说是至关重要的。许多研究人员建议使用关系型数据库存储和查询大型的 RDF 数据集。这种被称作关系型 RDF 数据库或者关系型 RDF 商店的系统现在正被频繁地应用在产品中。最近，在云计算中广泛使用的分布式技术如 Hadoop 何 Hbase 等，正用来开发分布式的和可扩展的 RDF 数据管理。据我们所知，这项工作提供了在 Hbase 和 MysQL 集群中使用我们的设计和算法解决方案来进行存储和查询 RDF 数据的关键性能的比较。

本篇论文的主要贡献有：(i) 用来在 HBase 中存储 RDF 数据的新型数据库结构设计（ii) 为由我们设计的模式在 HBase 中进行 SPARQL 三元组和基本图形的模式匹配提供了有效的算法（iii）高效的 SPARQL 和 SQL 之间转换的算法，它能使 MySQL 集群上进行的 SQL 查询变得平滑 (iv)HBase 和 MySQL 集群上存储和查询语义 Web 数据的高效和可扩展性的富有经验性的比较。我们的工作展示了在查询评估中有趣的模式，这表明了我们的算法是有希望的，并且暗示了云计算对于可扩展的语义 Web 的数据管理来说有很大的潜力。

本篇论文的组织如下：在第二章介绍了相关的工作，第三章和第四章介绍了我们对于 HBase 和 MySQL 集群上进行分布式 RDF 数据的存储和查询的设计和算法，第五章介绍了这两种方法在 the Third Provenance Challenge 和 Lehigh 大学进行的基准数据集的性能测试报告。最后我们在第六章进行了总结。

相关工作

除了作为 google 的 Bigtable 开源实现的 HBase，还有很多在 Apache 基金会下的项目，他们将焦点地方在分布式的计算，这些项目包括 Hadoop，Cassandra，Hive，Pig 和 CouchDB。Hadoop 实现了 MapReduce 软件架构和分布式文件系统。Cassandra 将一个完全分布式的设计和面向列的存储融合起来，并且将 MapReduce 作为支持的特性之一。Hive 在 Hadoop 之上处理数据仓库并且提供了他自己的查询语言 HQL。Pig 面向于使用它的高层的 Pig 拉丁语言来书写数据分析程序，这些程序最终被转化为 MapReduce 工作，以此完成分析大型的数据集。CouchDB 是一个分布式的、面向文档的非关系型数据库，它支持 JavaScript 编写的增量性的 MapReduce 查询。在学术界和工业界的其他项目，包括 Cheetah, Hadoop++, G-Store 和 HadoopDB 都有类似的思路。

接下来简要讨论一下关于分布式 RDF 数据管理的一些相关工作。参考文献 3 和 6 里呈现了评估 SPARQL 的基本图形模式的技术。参考文献 7 和 8 分别提出了在以 MapReduce 系统中分析型查询处理和 RDF 图形分布式推理的有效方法。参考文献 9 和 10 研究了在点对点的环境下 RDF 查询处理，11 和 12 报告了分布式 RDF 源之间联合查询的调解技术。13 介绍了在文本索引中 HBase 的用处。在参考文献 14 中虽然 S 声明了 spider 系统使用了 HBase 来处理 RDF 查询处理和扩展的 HBase，但是并没有详细的报告。最后，我们之前的工作展示了在 HBase 中 RDF 数据管理的最初发现。这篇论文提出了更新更有效的 HBase 表结构设计，更高效的 SPARQL 三元组和基本图形模式匹配和算法和对于分布式关系型 RDF 数据库富有经验的比较。我们的实验比较结果呈现出对于一些查询的数个数量级的提高，以及可扩展性的重大改进。这篇论文和我们之前的论文是关于 HBase 中语义 Web 数据管理最先发表的研究成果。我们对于 HBase 和

MySQL 集群的 RDF 数据管理技术的比较也是独一无二的。

在 Hbase 中的分布式 RDF 数据存储和查询

Hbase 将数据存储在表中，并且数据可以表述为非常稀疏的多维的排序的映射，这和传统的关系型数据库里的关系非常不同。一个 Hbase 里的表存储了根据 rowkey 排序的数据行。每一行有一个唯一的 rowkey 和任意数量的列，这样在两个不同行的列就不用一模一样。一个列的完整的名称包括一个列族和一个列修饰符，这里的列族是在表建立时就指定的，列族的数量是不会变的，但是列修饰符却可以动态的增加和删除。一个给定行的一列，我们称之为表单元格，能存储一对时间戳值，时间戳在单元格里是惟一的，并且值可以重复。表中的行在 Hbase 集群不同的机器上可以成为分布式的，并且能进行两个基本的操作：1. 表的扫描 2. 根据一个给定的 rowkey 或者列、时间戳等等来检索表格的数据。考虑到对于大数量级来说表格扫描访问路径是非常低效的，以 rowkey 来检索是最有效的方法。

表格稀疏的特性使得它们称为 RDF 数据非常富有吸引力的存储选择。RDF 图表通常也是很稀松的：不同的资源用不同的属性来注释，并且一些注解可能由于推理的缘故不会显示的指出。为了支持在 Hbase 表中有效的检索 RDF 数据，应该考虑到 SPARQL 构造的基本查询，比如三元模式等等。数据库最起码应该支持 RDF 三元组的主体、谓词、对象的值的检索。

我们建议使用两个表的数据库模式来存储 RDF 三元组，正如图 2 所示。sp 表存储了三元组的主体作为 rowkey，三元组的谓词作为列名字，三元组的对象作为单元格的值。op 表存储三元组的对象作为 rowkey，三元组的谓词作为列名字，三元组的主体作为单元格的值。表 2 显示了使用我们 RDF 三元组样例来存储这些表的二维的图形化展示。在这幅图里，s 和 o 代表了 rowkey 而不是列；类型，名字，属于什么组是列修饰符，这些修饰符属于共同的列族 p；代表了省略了时间戳的单元格值的集合。更加确切的说，行的这种表结构可以用 JavaScript 对象符号来表示为：

```
//the first row of Tsp
<C>:{
  p:{
    type: {t1: <Student> },
    name: {t2: "Carig" },
    memberof:{t3: <IEEE>}
 }
}
//the first row of Top
<Student>:{
 p: {
    type:{ t4:<C>,t5:<S>}
    }
}
```

RDF 数据建议的模式要求数据本身被存储两次以便用来保证系统的健壮性。表 Tsp 和 Top 分别可以根据一致的主体或对象来有效的检索三元组信息。而基于一个谓词的值的检索，由于他需要一个表的全表扫描，因而可能不会非常有效。为了解决这个问题，我们可以创建表 Tps 或 Tpo，将谓词放在 rowkey 的位置上，而将主体或对象作为列。然而，这样的解决方法只能提供轻微的改善，因为本体的谓词数量通常是比较固

定的且相对很小的，这暗示了新的表只能包含大量行的一小部分，而对于任何单一行的检索仍然是代价昂贵的。

为了使 Hbase 能够评估 SPARQL 查询，我们设计了三个函数来处理三元组模式和基本图表模式。

我们的第一个函数叫做 matchTP-T，是一个通用的函数，它既不依赖于我们的存储模式也不依赖于 Hbase。matchTP-T 将一个三元模式 tp 和一个三元组 t 作为输入，如果他们匹配则返回 true，否则返回 false。它的伪代码在 4 中呈现。

算法 1 中的函数 matchTP-DB 是用来根据我们的两个表的存储模式来匹配 HBase 数据库 DB 中的一个三元组模式 tp。这个函数的输出是一个多重集合 B，它包含了数据库中所有匹配的三元组。这个算法处理三个不相关的例子。首先，如果 tp 的主体模式不是一个变量，这个函数从 Tsp 表中检索出结果。如果 tp.pp 不是变量，只有拥有了列修饰符 tp.pp 的列的值从行中检索出来。否则，所有的列都将被检索。由于 tp.op 可能是也可能不是一个变量，matchTP-T 被用在所有的三元组上以除去不匹配的组。在这个过滤之后，B 中的组被返回。第二，如果 tp 的对象模式不是一个变量，这个函数使用相似的方法从表 Top 中检索出数据。最后，当 tp.sp 和 tp.op 都是变量时，其中的一个表必须扫描来检索出所有的行。如果 tp.pp 不是一个变量，不匹配的列将被舍弃，否则，所有列中的值都将被使用。

我们最后一个函数 matchBGP-DB 在算法 2 中给出轮廓。它将包含了一系列三元组 tp1,tp2... 的 SPARQL 基本的图表模式 bgp 和 HBase 数据库进行比较，并且返回一个由匹配的三元组组成的图表集合 B。这个算法首先使用两个准则来排序 bgp 中的三元组：（1）为了减少迭代的次数首先应将产出结果小的三元模式放在首位，（2）为了避免不必要的笛卡尔积应将那些有共同变量的三元组优先考虑。作为一个例子，考虑下面的查询和它的排序版本。原来查询中的顺序并不符合所需的条件：tp1 在数据集里产生了大量的所有大学里的学生的结果集；tp2 和 tp1 没有公告变量，并且在 tp1 和 tp2 之间必须计算耗费内存的笛卡儿积；被记录的查询既能节省内存又能节省网络传输时间：对于三元组 tp3, 它不仅仅因为有最小结果集它被放在第一位，而且笛卡尔积也被移除了。

接下来，这个算法使用 matchTP-DB 算法来评估 bgp 里的排好序的第一个三元组。如果 B 中的结果为空，这个算法就不再评估它的子三元组而返回一个空结果集。否则，matchBGP-DB 函数就会迭代地计算三元组或者公共变量或者没有公共变量时计算笛卡尔积。每个 join 连接就像关系型数据库中的连接策略。公共变量首先被 B 集合中绑定的替换，并使用 matchTP-DB 函数评估 TP 集合中的三元组 tp’。如果 tp’ 生成了非空的结果，B’ 中的三元组就将 B 中相应的三元组串联在一起。

在 Mysql 集群中分布式 RDF 数据存储和查询

关系型 RDF 数据库使用几种数据模式产生的方法，包括 schema-oblivious,schema-aware 等等。这些方法标志了不同的数据库关系，比如属性，类，类的主体和客体及集群化的数据库表等等。在这项工作里，我们使用 schema-oblivious 方法作用到单个表 T(s,p,o) 上，这里的列 s,p,o 相应的存储了三元组的主体，谓词，对象。

我们选择这一个模式是由于三种原因。首先，它可以支持无模式的修改本体演化。Hbase 建议的模式也是非常灵活的，因为只有列的修饰符才会动态的变化而且是在行级

上进行变化的。第二，关系型 RDF 数据库中涉及到的大部分表都可以看作是表水平分区的结果。然而，分区的工作已经被 Mysql 集群自动的进行了。最后，这个模式允许无损存储，并且很容易实现，特别是它大大的简化了 SPARQL 到 SQL 的转换工作，这个工作需要查询存储的 RDF 数据。

为了在 Mysql 集群的数据库架构上执行 SPARQL 查询，我们提出了一个基本图表模式的 SPARQL 到 SQL 的查询转换算法。该算法是以我们以前的工作 15 为基础，但是它又进行了优化，以生成平坦的 SQL 查询。平坦的查询会移除一系列在基本图表模式上重新排序的三元组，因为关系型的查询优化器足以自动地选出一个好的连接查询执行序列。

性能研究

这一章讨论了在 HBase 和 Mysql 集群上进行分布式存储和查询 RDF 数据的不同方法的经验性比较。A. 实验前提硬件。我们的实验使用 9 个有着标准硬件的商业机器。每个机器有 3GHz 64bit 的奔腾四核处理器，2GB DDR2-533RAM，80GB 7200 转串行 ATA 硬件驱动。机器通过网络连接到一个戴尔巨型以太网交换机，该交换机上安装了 64 位的 Debian Linux 和 Oracle JDK6。HBase 和 Mysql 集群。使用了带有更改内核库的 Hadoop0.20.2，还有 HBase0.90. 为了稳定性我们对默认的配置做了细微的更改，包括设置每个数据块复制两次，增加 HBase 最大堆大小到 1.2GB。同时使用了基于快速入门指南配置上更改的 Mysql 集群 7.1.9a，将 NDB 数据节点使用的内存增加。我们的实现。我们的算法使用 Java 语言来实现，并且实验通过使用 Bash 脚本来自动地和可重复地执行 Java 类文件和存储的结果。查询性能评估。Hbase 和 Mysql 集群在 PC3 和 LUBM 数据集上的查询性能和可扩展性结果在图 4 上报告。PC3 基准测试送三个不同复杂性的查询：Q1 是最简单的查询只用了一个三元组。Q2 有三个三元模式，Q3 是最复杂的，包括 6 个模式。所有三个查询里基本的图表模式返回一个小量结果。Hbase 和 Mysql 集群都显示出非常有效的响应时间，而前者稍微的快一点。在第六天，Hbase 之前保持的持平的时间也有了轻微的上升，这意味着图有一个小的斜坡（数据集的大小增加了近 10 倍，而响应时间只有大约 2 4 倍的增加），类似的行为也在 LUBM 查询中观察到。

外文译文原文

Distributed Semantic Web Data Management in HBase and MySQL Cluster

Craig Franke, Samuel Morin, Artem Chebotko †, John Abraham, and Pearl Brazier
Department of Computer Science
University of Texas - Pan American
1201 West University Drive, Edinburg, TX 78539-2999, USA
† *Corresponding author. Email: artem@cs.panam.edu*

***Abstract*—Various computing and data resources on the Web are being enhanced with machine-interpretable semantic descriptions to facilitate better search, discovery and integration. This interconnected metadata constitutes the Semantic Web, whose volume can potentially grow the scale of the Web. Efficient management of Semantic Web data, expressed using the W3C's Resource Description Framework (RDF), is crucial for supporting new data-intensive, semantics-enabled applications. In this work, we study and compare two approaches to distributed RDF data management based on emerging cloud computing technologies and traditional relational database clustering technologies. In particular, we design distributed RDF data storage and querying schemes for HBase and MySQL Cluster and conduct an empirical comparison of these approaches on a cluster of commodity machines using datasets and queries from the Third Provenance Challenge and Lehigh University Benchmark. Our study reveals interesting patterns in query evaluation, shows that our algorithms are promising, and suggests that cloud computing has a great potential for scalable Semantic Web data management.**

***Keywords*-Semantic Web; cloud computing; distributed database; SPARQL; SQL; RDF; query; performance; scalability; HBase; MySQL Cluster**

I. Introduction

The World Wide Web Consortium (W3C) has recommended and standardized a number of principles, languages, frameworks and best practices to interconnect various metadata into a next-generation web – the Semantic Web. The W3C's metadata acquisition languages include Resource Description Framework (RDF), RDF in attributes (RDFa), RDF Schema (RDFS), and Web Ontology Language (OWL). Government, academia, and industry actively embrace these technologies for capturing and sharing metadata on the Semantic Web. Just to name a few examples, oeGOV is making and publishing OWL ontologies for e-Government, U.S. census data is being published in RDF, bioinformaticians maintain the Universal Protein Resource (UniProt) in RDF, geoscientists publish worldwide geographical RDF database GeoNames, the largest electronics retailer in the U.S., BestBuy, publishes its full catalog in RDF, the largest social networking provider in the U.S., Facebook, embeds metadata in its webpages using RDFa, and the services computing community enhances existing Web services with semantic annotations using vocabularies, such as Semantic Markup for Web Services (OWL-S), Web Service Semantics (WSDL-S), and Semantic Web Services Ontology (SWSO).

The RDF data model is a directed, labeled graph that can also be serialized and viewed as a set of triples. A running example in this paper includes 10 triples that describe the authors using the Lehigh University Benchmark (LUBM) vocabulary [1] as shown in Fig. 1. Each triple consists of a subject, predicate, and object and defines a relationship between a subject and an object. In the figure, <> and "" denote resource identifiers and literals of some data type, respectively. For example, the first three triples state that a resource with identifier *C* is a *Student*, has name *Craig* and is a member of *IEEE*. This sample dataset can be queried using SPARQL – a standard query language for RDF. SPARQL uses triple patterns and graph patterns that are matched over RDF data. For example, query Q14 from LUBM contains one triple pattern *?X <type> <UndergraduateStudent>* that returns all undergraduate student identifiers as bindings of variable *?X*. More details on SPARQL features and semantics can be found in the W3C's SPARQL specification.

With the rapid growth of the Semantic Web and widespread use of RDF as the primary language for metadata, efficient management of RDF data will become crucial for supporting new semantics-enabled applications in various domains. Many researchers have proposed using relational databases to store and query large RDF datasets. Such systems, called relational RDF databases or relational RDF stores [2], are now frequently in production. More recently, distributed technologies that are often used in cloud computing, such as Hadoop[1] and HBase[2], are being explored for distributed and scalable RDF data management [3], [4]. To our best knowledge, this work provides the first performance comparison of the two worlds using our design and algorithmic solutions for storing and querying RDF data in HBase and MySQL Cluster.

The main contributions of this paper are: (i) a novel database schema design for storing RDF data in HBase, (ii) efficient algorithms for SPARQL triple and basic graph pattern matching in HBase according to our schema, (iii) efficient SPARQL-to-SQL translation algorithm that results

[1] Apache Hadoop, http://hadoop.apache.org
[2] Apache HBase, http://hbase.apache.org

```
<C>     <type>      <Student>
<C>     <name>       "Craig"
<C>   <memberOf>     <IEEE>
<S>     <type>      <Student>
<S>     <name>        "Sam"
<S>   <memberOf>     <ACM>
<A>     <type>      <Faculty>
<A>     <name>       "Artem"
<A>   <memberOf>     <IEEE>
<A>   <memberOf>     <ACM>
```

Figure 1. Sample RDF triples.

in flat SQL queries over our schema in MySQL Cluster, and (iv) empirical comparison of the proposed HBase and MySQL Cluster approaches for efficient and scalable storing and querying of Semantic Web data. Our work reveals interesting patterns in query evaluation, shows that our algorithms are promising, and suggests that cloud computing has a great potential for scalable Semantic Web data management.

The organization of this paper is as follows. Related work is discussed in Section II. Our design and algorithms for distributed RDF data storage and querying in HBase and MySQL Cluster are presented in Sections III and IV, respectively. The performance study of the two approaches using datasets and queries from the Third Provenance Challenge and Lehigh University Benchmark is reported in Section V. Finally, our concluding remarks are given in Section VI.

II. Related Work

Besides HBase, which is an open-source implementation of Google's Bigtable [5], there are multiple projects under the Apache umbrella that focus on distributed computing, including Hadoop, Cassandra, Hive, Pig, and CouchDB. Hadoop implements a MapReduce software framework and a distributed file system. Cassandra blends a fully distributed design with a column-oriented storage model and supports MapReduce as one of its features. Hive deals with data warehousing on top of Hadoop and provides its own Hive QL query language. Pig is geared towards analyzing large datasets through use of its high-level Pig Latin language for expressing data analysis programs, which are then turned into MapReduce jobs. CouchDB is a distributed, document-oriented, non-relational database that supports incremental MapReduce queries written in JavaScript. Along the same lines, other projects in academia and industry include Cheetah, Hadoop++, G-Store, and HadoopDB.

Several related works on distributed RDF data management are briefly discussed in the following. Techniques for evaluating SPARQL basic graph patterns using MapReduce are presented in [3] and [6]. Efficient approaches to analytical query processing and distributed reasoning on RDF graphs in MapReduce-based systems are proposed in [7] and [8], respectively. RDF query processing in peer-to-peer environments is studied in [9] and [10], and mediation techniques for federated querying of distributed RDF sources are reported in [11] and [12]. Use of HBase for text indexing is described in [13]. While the SPIDER system [14] that uses HBase for RDF query processing and the HBase extension for Jena[3] are announced, no details are reported. Finally, our previous work [4] presents our initial findings on RDF data management in HBase. This paper, when compared to [4], proposes new, more effective HBase database schema design, more efficient algorithms for SPARQL triple and basic graph pattern matching, and an empirical comparison with a distributed relational RDF database. Our experimental comparison with [4] (not reported in the paper) showed several orders of magnitude speedup for some queries and substantial improvements in scalability. To our best knowledge, this paper and our previous paper [4] are the first published research works on Semantic Web data management in HBase. Our comparison of RDF data management techniques in HBase and MySQL Cluster is also unique.

III. Distributed RDF Data Storage and Querying in HBase

HBase stores data in tables that can be described as sparse multidimensional sorted maps and are structurally different from relations found in conventional relational databases. An HBase table (hereafter "table" for short) stores data rows that are sorted based on the row keys. Each row has a unique row key and an arbitrary number of columns, such that columns in two distinct rows do not have to be the same. A full column name (hereafter "column" for short) consists of a column family and a column qualifier (e.g., *family:qualifier*), where column families are usually specified at the time of table creation and their number does not change and column qualifiers are dynamically added or deleted as needed. A column of a given row, which we denote as table cell, can store a list of timestamp-value pairs, where timestamps are unique in the cell scope and values may contain duplicates. Rows in a table can be distributed over different machines in an HBase cluster and searched using two basic operations: (1) table scan and (2) retrieval of row data based on a given row key and, if available, columns and timestamps. Given that the table scan access path is inefficient for large datasets, the row key-based retrieval is the best feasible choice.

The sparse nature of tables makes them an attractive storage alternative for RDF data. RDF graphs are usually sparse as well: different resources are annotated with different properties and some annotations may not be stated explicitly due to inference. To support efficient retrieval of RDF data from tables in HBase, the basic querying constructs of SPARQL, such as triple patterns, should be considered. At the very minimum, the database should support retrieval of RDF triples based on values of their subjects, predicates, objects, and their arbitrary combination.

We propose to use a database schema with two tables to store RDF triples as shown in Fig. 2. Table T_{sp} stores triple

[3]HBase Graph for Jena, http://cs.utdallas.edu/semanticweb/HBase-Extension/hbase-extension.html

T_{sp}

s	p:type	p:name	p:memberOf	...
<C>	{<Student>}	{"Craig"}	{<IEEE>}	...
<S>	{<Student>}	{"Sam"}	{<ACM>}	...
<A>	{<Faculty>}	{"Artem"}	{<IEEE>,<ACM>}	...

T_{op}

o	p:type	p:name	p:memberOf	...
<Student>	{<C>,<S>}			...
<Faculty>	{<A>}			...
"Craig"		{<C>}		...
"Sam"		{<S>}		...
"Artem"		{<A>}		...
<IEEE>			{<C>,<A>}	...
<ACM>			{<S>,<A>}	...

Figure 2. Storage schema and sample instance in HBase.

subjects as row keys, triple predicates as column names and triple objects as cell values. Table T_{op} stores triple objects as row keys, triple predicates as column names and triple subjects as cell values. Fig. 2 shows a two-dimensional graphical representation of these tables with our sample RDF triples (see Fig. 1) stored. In the figure, s and o denote row keys rather than columns; *type*, *name*, and *memberOf* are column qualifiers that belong to the same column family p; { } denote sets of cell values with timestamps omitted. More precisely, the structure of the rows can be shown using JavaScript Object Notation (JSON):

```
//the first row of T_sp
<C>: {
 p: {
   type:      { t1: <Student> },
   name:      { t2: "Craig" },
   memberOf:  { t3: <IEEE> }
 }
}
//the first row of T_op
<Student>: {
 p: {
   type:      { t4: <C>,
                t5: <S> }
 }
}
```

In the first row of T_{sp}, $<C>$ is a row key, p is a column family, *type*, *name*, and *memberOf* are column qualifiers, t_1, t_2, and t_3 are timestamps, and the rest are values. The structure of the first row of T_{op} can be interpreted in a similar way but it should be noted that, while the graphical representation in Fig. 2 shows blank values for some table cells, the row contains no information about such values or the respective columns. This illustrates the sparse storage nature of HBase tables and shows that no space is wasted.

The proposed schema requires that RDF data is stored twice - replication that contributes to the robustness of the system. Tables T_{sp} and T_{op} can be used to efficiently retrieve triples with known subjects and objects, respectively. Retrieval of triples based on a predicate value requires a scan of one of the tables, which may not be efficient. To try to remedy this problem, we could have created a table, i.e., T_{ps} or T_{po}, with predicates as row keys and subjects or objects as columns. However, such a solution can only provide marginal improvements, since the number of predicates in an ontology is usually fixed and relatively small, which implies that this new table can contains only a small number of large rows (one per distinct predicate) and retrieval of any individual row is still expensive.

For HBase to be able to evaluate SPARQL queries, we design three functions that deal with triple patterns and basic graph patterns.

Our first function, matchTP-T, is a general-purpose function that depends on neither our storage schema nor HBase. matchTP-T takes a triple pattern tp and a triple t and returns *true* if they match or *false* otherwise. Its pseudocode is outlined in [4].

Function matchTP-DB as outlined in Algorithm 1 is used to match a triple pattern tp in an HBase database DB according to our storage schema with two tables. The output of this function is a bag (multi-set) B that holds all matching triples in the database. The algorithm deals with three disjoint cases. First, if tp's subject pattern is not a variable, the function retrieves matching triples from table T_{sp}, such that a row with key $tp.sp$ is accessed. If $tp.pp$ is not a variable, only values in the column with qualifier $tp.pp$ are retrieved for this row; otherwise, all columns must be retrieved. Triples are reconstructed from row keys, column qualifiers, and cell values and are placed into B. Since $tp.op$ may not be a variable or it may be a variable that occurs twice in the triple pattern, matchTP-T is applied on all the triples to eliminate non-matching ones. After this filtering, triples in B are returned. Second, if tp's object pattern is not a variable, the function retrieves matching triples from table T_{op} using a similar strategy. Finally, when both $tp.sp$ and $tp.op$ are variables, one of the tables must be scanned to retrieve all rows. If $tp.pp$ is not a variable, non-matching columns are discarded; otherwise, values in all columns are used.

Our last function matchBGP-DB is outlined in Algorithm 2. It matches a SPARQL basic graph pattern bgp that consists of a set of triple patterns tp_1, tp_2, ..., tp_n over an HBase database and returns a relation with a bag B of graphs constituted by matching triples. The algorithm starts by ordering triple patterns in bgp using two criteria: (1) triple patterns that yield a smaller result should be evaluated first to decrease a number of iterations and (2) triple patterns that have a shared variable with preceding triple patterns should be given a preference over triple patterns with no shared variables to avoid unnecessary Cartesian products. As an example, consider the following query from LUBM [1] and its reordered version:

```
//original query Q7 from LUBM
tp1: ?X <type> <Student> .
tp2: ?Y <type> <Course> .
tp3: <http://...Professor0> <teacherOf> ?Y .
tp4: ?X <takesCourse> ?Y .
//reordered basic graph pattern
tp3: <http://...Professor0> <teacherOf> ?Y .
tp2: ?Y <type> <Course> .
tp4: ?X <takesCourse> ?Y .
tp1: ?X <type> <Student> .
```

Algorithm 1 Matching a triple pattern over a database

function matchTP-DB
input: triple pattern $tp = (sp, pp, op)$, database $DB = \{T_{sp}, T_{op}\}$
output: bag of triples $B_{(sp,pp,op)} = \{t | t \text{ is in } DB \wedge t \text{ matches } tp\}$
$B = \emptyset$
if $tp.sp$ is not a variable **then**
  if $tp.pp$ is not a variable **then**
    Retrieve triples into bag B from T_{sp} where row key $s = tp.sp$ using column $tp.pp$
  else
    Retrieve triples into bag B from T_{sp} where row key $s = tp.sp$ using all columns
  end if
  Remove any triple $t \in B$ from B if matchTP-T(tp, t) = *false*
  return B
end if
if $tp.op$ is not a variable **then**
  if $tp.pp$ is not a variable **then**
    Retrieve triples into bag B from T_{op} where row key $o = tp.op$ using column $tp.pp$
  else
    Retrieve triples into bag B from T_{op} where row key $o = tp.op$ using all columns
  end if
  Remove any triple $t \in B$ from B if matchTP-T(tp, t) = *false*
  return B
end if
if $tp.pp$ is not a variable **then**
  Retrieve triples into bag B from T_{sp} (or T_{op}) using column $tp.pp$
else
  Retrieve triples into bag B from T_{sp} (or T_{op}) using all columns
end if
Remove any triple $t \in B$ from B if matchTP-T(tp, t) = *false*
return B
end function

Algorithm 2 Matching a basic graph pattern over a database

function matchBGP-DB
input: basic graph pattern $bgp = \{tp_1, tp_2, \ldots, tp_{n-1}, tp_n\}$ and $n \geq 1$, database $DB = \{T_{sp}, T_{op}\}$
output: bag of tuples $B_{(tp_1.sp, tp_1.pp, tp_1.op, tp_2.sp, \ldots)} = \{g | g \text{ is a graph in } DB \wedge g \text{ matches } bgp\}$
$B = \emptyset$
Order triple patterns in bgp, such that triple patterns that yield a smaller result and triple patterns that have a shared variable with preceding triple patterns should be evaluated first.
Let ordered $bgp = (tp_1, tp_2, \ldots, tp_n)$
B = matchTP-DB(tp_1, DB)
if $B = \emptyset$ **then return** B **end if**
for each tp_i in $(tp_2, \ldots, tp_n)$ **do**
  if tp_i has shared variables with $tp_{i-1}, \ldots, tp_1$ **then**
    Let TP be a set of triple patterns obtained by substituting shared variables with their respective bindings from B
    for each tp' in TP **do**
      B' = matchTP-DB(tp', DB)
      if $B' \neq \emptyset$ **then**
        Add triples in B' to B by concatenating each triple $t' \in B'$ with every tuple $t \in B$ if t's bindings were used in variable substitution to obtain tp'
      else
        Remove any tuple t from B if t's bindings were used in variable substitution to obtain tp'
        if $B = \emptyset$ **then return** B **end if**
      end if
    end for
  else
    B' = matchTP-DB(tp_i, DB)
    Compute Cartesian product of B and B', i.e. $B = B \times B'$
  end if
end for
return B
end function

The order in the original query does not satisfy the desired criteria: tp_1 yields a large result set with all students across all universities in a dataset; tp_2 has no shared variables with tp_1 and a memory-expensive Cartesian product must be computed between tp_1's and tp_2's results. The reordered query can save both memory and network transfer time: not only is tp_3, the triple pattern with the smallest result, placed at the first position, but the Cartesian product is also eliminated.

Next, the algorithm evaluates the first triple pattern in ordered bgp using matchTP-DB. If the result in B is empty, the algorithm returns an empty result without evaluating subsequent triple patterns. Otherwise, matchBGP-DB iterates over other triple patterns computing either joins on shared variables or Cartesian products if no shared variables exist. Each join resembles the index-nested-loops join strategy known in relational databases. Instead of directly evaluating triple pattern tp_i using matchTP-DB, shared variables are first substituted with their bindings found in B and the resulting triple patterns tp' in set TP are evaluated using matchTP-DB. If tp' yields a non-empty result, triples in B' are concatenated with the corresponding triples in B; otherwise, previous solutions from B whose variable bindings were used in variable substitution to obtain tp' are removed as the join condition has failed.

Other SPARQL constructs, such as projection (*SELECT*), filtering (*FILTER*), alternative graph patterns (*UNION*), and optional graph patterns (*OPTIONAL*) can be incorporated in the presented algorithmic framework, but is out of this paper scope.

IV. Distributed RDF Data Storage and Querying in MySQL Cluster

Relational RDF databases use several approaches to database schema generation that include schema-oblivious, schema-aware, data-driven, and hybrid strategies [15]. These approaches feature various database relations, such as property, class, class-subject, class-object, and clustered property tables. In this work, we use a schema-oblivious approach that employs a generic schema with a single table $T(s, p, o)$, where columns s, p, and o store triple subjects, predicates, and objects, respectively. Fig. 3 shows table T with our sample RDF triples (see Fig. 1) stored.

Our rationale for choosing this schema is threefold. First, it can support ontology evolution with no schema modifications. The schema proposed for HBase is also very flexible as only column qualifiers may dynamically change and such changes are performed on the row level. Second, most mentioned tables employed by relational RDF databases can be viewed as a result of horizontal partitioning of table T. However, partitioning is already performed by MySQL Cluster automatically. Finally, this schema allows lossless storage and is easy to implement. In particular, it greatly simplifies SPARQL-to-SQL translation that is required to

T

s	p	o
<C>	<type>	<Student>
<C>	<name>	"Craig"
<C>	<memberOf>	<IEEE>
<S>	<type>	<Student>
<S>	<name>	"Sam"
<S>	<memberOf>	<ACM>
<A>	<type>	<Faculty>
<A>	<name>	"Artem"
<A>	<memberOf>	<IEEE>
<A>	<memberOf>	<ACM>

Figure 3. Storage schema and sample instance in MySQL Cluster.

query stored RDF data.

To execute SPARQL queries over our database schema in MySQL Cluster, we present a SPARQL-to-SQL query translation algorithm for basic graph patterns. The algorithm is based on our previous work [15] on semantics-preserving SPARQL-to-SQL translation, but it is optimized to generate flat SQL queries. Query flattening (vs. nesting) removes a concern of triple pattern reordering in basic graph patterns since a relational query optimizer is capable of selecting a "good" join execution order automatically.

Algorithm 3 Translation of SPARQL basic graph patterns to flat SQL queries

function BGPtoFlatSQL
input: basic graph pattern $bgp = \{tp_1, tp_2, \ldots, tp_{n-1}, tp_n\}$ and $n \geq 1$, database $DB = \{T\}$
output: flat SQL query
Assign a unique alias a_i to each triple pattern $tp_i \in bgp$
$select$ = ""; $from$ = ""; $where$ = ""
//Construct the SQL `From` clause:
for each $tp_i \in bgp$ **do**
  $from$ += "T $\$a_i$, "
end for
//Construct an inverted index (hash) h on variables in bgp:
for each $tp_i \in bgp$ **do**
  for each variable $?v$ found in tp_i **do**
    Let p be "s", "p", or "o" if $?v$ is at the subject, predicate, or object position, respectively, in tp_i
    $h(?v) = h(?v) \cup \{$"$\$a_i.\$p$"$\}$
  end for
end for
//Construct the SQL `Where` clause:
for each $tp_i \in bgp$ **do**
  for each instance or literal l found in tp_i **do**
    Let p be "s", "p", or "o" if l is at the subject, predicate, or object position, respectively, in tp_i
    $where$ += "$\$a_i.\$p$ = '$\$l$' `And` "
  end for
end for
for each distinct variable $?v$ found in bgp and $|h(?v)| > 1$ **do**
  Let $x \in h(?v)$
  for each $y \in h(?v)$ and $y \neq x$ **do**
    $where$ += "$\$x = \$y$ `And` "
  end for
end for
//Construct the SQL `Select` clause:
for each distinct variable $?v$ found in bgp **do**
  Let $x \in h(?v)$
  Let m is the name of variable $?v$
  $select$ += "$\$x$ `As` $\$m$, "
end for
return "`Select` $\$select$ `From` $\$from$ `Where` $\$where$"
end function

The BGPtoFlatSQL function is outlined in Algorithm 3. It translates a SPARQL basic graph pattern bgp that consists of a set of triple patterns tp_1, tp_2, ..., tp_n into an equivalent flat SQL query that can be executed over a MySQL Cluster database with our schema. BGPtoFlatSQL constructs $from$, $where$, and $select$ clauses of an SQL query as follows. For each triple pattern in bgp, a unique table alias is assigned and table T with this alias is appended to the $from$ clause. The algorithm then computes an inverted index on all variables in bgp, such that each distinct variable is associated with attributes in the respective tables from the $from$ clause. The $where$ clause is first constructed to ensure that any non-variables in bgp are restricted to their values (e.g., literals or identifiers). The inverted index is then used to append join conditions into the $where$ clause, such that all attributes that correspond to the same variable must be equal. Finally, the $select$ clause is generated to include attributes that correspond to every distinct variable in bgp, with attributes being renamed as variable names. The following example illustrates the result of a translation performed with BGPtoFlatSQL:

```
//input SPARQL query Q7 from LUBM
tp1: ?X <type> <Student> .
tp2: ?Y <type> <Course> .
tp3: <http://...Professor0> <teacherOf> ?Y .
tp4: ?X <takesCourse> ?Y .
//output equivalent SQL query
Select tp1.s As X, tp2.s As Y
From T tp1, T tp2, T tp3, T tp4
Where tp1.p = '<type>' And
      tp1.o = '<Student>' And
      tp2.p = '<type>' And
      tp2.o = '<Course>' And
      tp3.s = '<http://...Professor0>' And
      tp3.p = '<teacherOf>' And
      tp4.p = '<takesCourse>' And
      tp1.s = tp4.s And tp2.s = tp3.o And
      tp2.s = tp4.o
```

Translation of other SPARQL constructs into SQL is out of this paper scope; details can be found in [15].

V. Performance Study

This section reports our empirical comparison of the proposed approaches to distributed Semantic Web data storage and querying in HBase and MySQL Cluster.

A. Experimental Setup

Hardware. Our experiments used nine commodity machines with identical hardware. Each machine had a late-model 3.0 GHz 64-bit Pentium 4 processor, 2 GB DDR2-533 RAM, 80 GB 7200 rpm Serial ATA hard drive. The machines were networked together via their add-on gigabit Ethernet adapters connected to a Dell PowerConnect 2724 gigabit Ethernet switch and were all running 64-bit Debian Linux 5.0.7 and Oracle JDK 6.

HBase and MySQL Cluster. Hadoop 0.20.2, with a modified core library, and HBase 0.90 were used. Minor changes to the default configuration for stability included setting each block of data to replicate two times and increasing the HBase max heap size to 1.2 GB. MySQL Cluster 7.1.9a was used with a modified configuration based on the MySQL Cluster

Quick Start Guide with increased memory available for use by NDB data nodes.

Our implementation. Our algorithms were implemented in Java and the experiments were conducted using Bash shell scripts to execute the Java class files and store the results in an automated and repeatable manner.

B. Datasets and Queries

The experiments used datasets from the Third Provenance Challenge (PC3)[4] and Lehigh University Benchmark (LUBM) [1]. PC3 employed the Load Workflow that was a variation of a workflow used in the Pan-STARRS project. Via simulation, a number of scientific workflow provenance documents for multiple workflow runs was generated and represented using Tupelo's OWL vocabulary available from the Open Provenance Model website[5]. Each workflow execution generated approximately 700 RDF triples. Table I indicates the characteristics of each PC3 dataset. The three PC3 SPARQL queries utilized for the experiments can be found in our previous work [4]. LUBM is a popular benchmark for RDF databases that includes the OWL university ontology, RDF data generator, and 14 test queries. Table II indicates the characteristics of each generated LUBM dataset. The LUBM queries expressed in a KIF-like language can be found on the LUBM website[6]; for the purpose of our experiments, they were rewritten in SPARQL. Since our experiments tested query performance and not reasoning ability, each generated LUBM dataset was augmented with additional triples needed to produce the sample query results supplied by LUBM.

Table I
PC3 DATASET CHARACTERISTICS.

Dataset	# of workflow runs	# of RDF triples	Disk space
$D1$	1	700	86 KB
$D2$	10	7,000	860 KB
$D3$	100	70,000	8.7 MB
$D4$	1,000	700,000	88 MB
$D5$	10,000	7,000,000	895 MB
$D6$	100,000	70,000,000	9 GB

C. Data Ingest Performance

Due to the space limit, we only report a few observations on data ingest. First, out of tested statement-by-statement, batch, and bulk load methods, MySQL Cluster and HBase showed the best data ingest performance with bulk and batch methods, respectively. Second, MySQL Cluster was able to bulk load datasets up to $D5$ and $L8$ and HBase successfully batch loaded all the datasets. Finally, MySQL Cluster initially demonstrated a significant advantage over HBase (3 times faster on $L1$), however this performance advantage decreased with dataset size growth (only 1.5 times faster on $L8$); it also should be noted that HBase required to store twice as many triples as MySQL Cluster.

Table II
LUBM DATASET CHARACTERISTICS.

Dataset	# of universities	# of RDF triples	Disk space
$L1$	1	38,600	4.4 MB
$L2$	5	563,000	68 MB
$L3$	10	1,211,000	146 MB
$L4$	30	3,908,000	477 MB
$L5$	50	6,593,000	807 MB
$L6$	70	9,308,000	1.1 GB
$L7$	90	11,964,000	1.5 GB
$L8$	110	14,649,000	1.8 GB
$L9$	200	26,635,000	3.3 GB
$L10$	400	53,301,000	6.6 GB
$L11$	600	80,043,000	9.9 GB

D. Query Evaluation Performance

HBase and MySQL Cluster query performance and scalability on PC3 and LUBM datasets are reported in Fig. 4. The PC3 benchmark used three queries with varying complexity: $Q1$ was the simplest query with one triple pattern, $Q2$ had three triple patterns, and $Q3$ was the most complex one consisting of six triple patterns. The basic graph patterns in all three queries returned a small result. Both HBase and MySQL Cluster showed very efficient and comparable response times, with the former being slightly faster. At $D6$, HBase took a slight upward turn in times that had previously remained nearly flat, which signifies that the graphs have a small slope (while the dataset size increased by a factor of 10, the response times increased by a factor of only around 2 to 4); similar behavior was also observed for some LUBM queries.

The LUBM benchmark used 14 queries whose complexities are shown in Table III. LUBM query evaluation results for HBase and MySQL Cluster revealed several interesting patterns, denoted as *A*, *B*, *C*, *D*, and *E* in Table III. Pattern *A* ($Q1$, $Q3$, $Q7$, and $Q11$) is characterized by the rapidly increasing query execution time for MySQL Cluster and nearly constant response time for HBase as the dataset size increased. Pattern *B* ($Q2$ and $Q14$) is characterized by rapid performance degradation in both systems. While $Q2$ had six triple patterns, $Q14$ had only one triple pattern that retrieved all undergraduate students across all universities in the database. Both queries yielded large results, such that results for $L9$, $L10$, and $L11$ could not fit into main memory on the HBase master server. In the case of $Q14$, which involved no joins, it is evident that the major factor in query performance is data transfer time and it is hardly possible to achieve better performance on the given hardware. Patterns *C* ($Q4$, $Q5$, $Q6$, and $Q8$) and *D* ($Q9$, $Q10$, and $Q13$) include queries whose performance showed limited or no growth in execution times with an increase in the data size in both systems. Pattern *C* queries were approximately 2 to 3 times faster on MySQL Cluster and pattern *D* queries

[4]Third Provenance Challenge, http://twiki.ipaw.info/bin/view/Challenge/ThirdProvenanceChallenge

[5]Open Provenance Model, http://openprovenance.org

[6]Lehigh University Benchmark, http://swat.cse.lehigh.edu/projects/lubm/

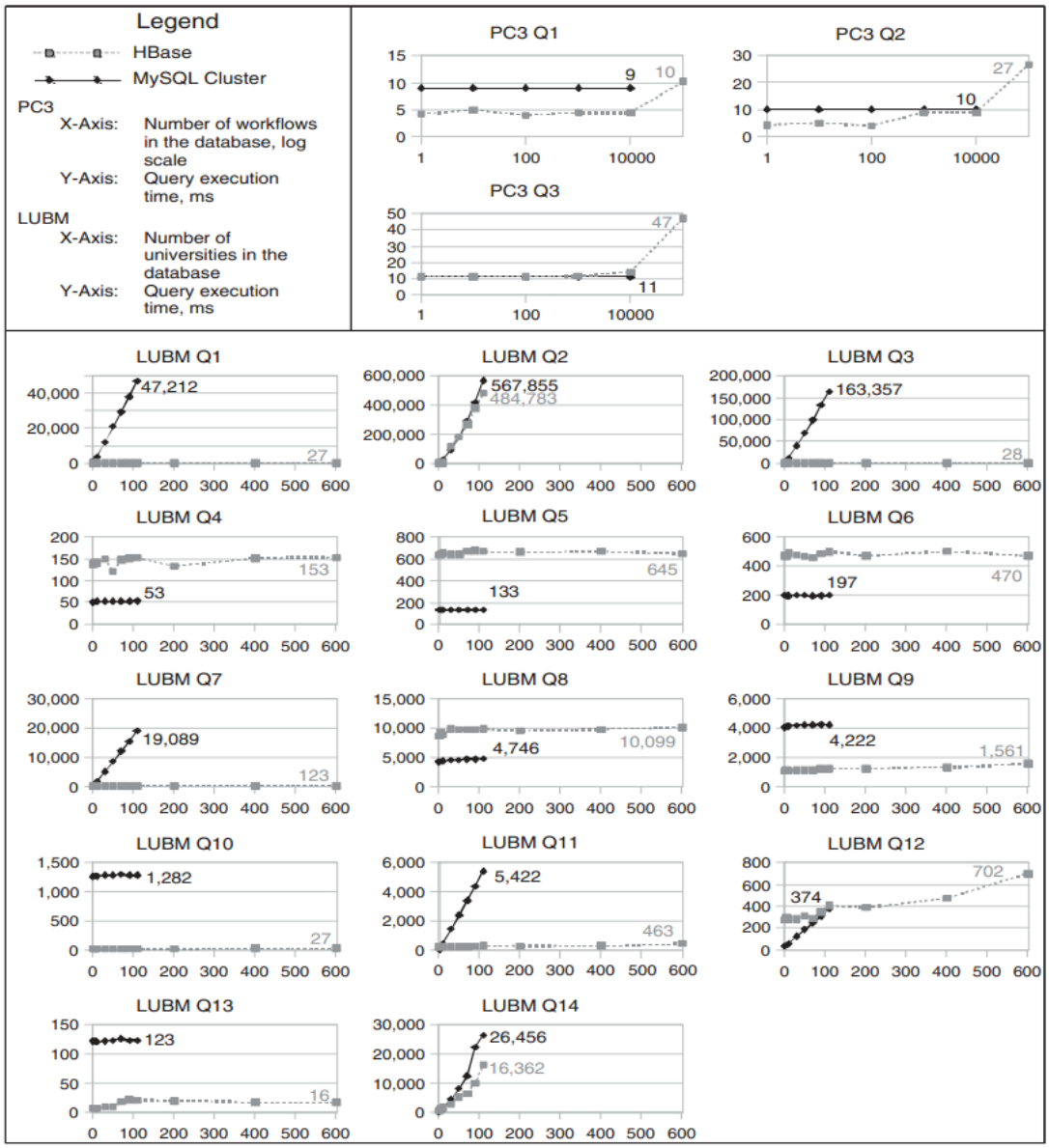


Figure 4. Query performance and scalability.

anywhere from 3 to 47 times faster on HBase. Pattern nds out on its own with a single representative query - For smaller datasets, $Q12$ was much faster on MySQL er, however its performance quickly decreased on larger ets, much like in pattern A. HBase, on the other hand, onstrated a gradual increase in execution time: close e 100 university mark, HBase performance exceeded QL Cluster performance.

e comparison of the query evaluation patterns and complexity in LUBM (see Table III) does not reveal any strong correlation between the two characteristics. The query complexity is not the sole indicator of query performance under HBase and MySQL Cluster: the size of intermediate and final results can have a significant impact.

Overall, in our experiments, the HBase approach showed better performance and scalability than the MySQL Cluster approach. Neglecting $Q2$ and $Q14$ of LUBM, which are expensive due to returning large results, the evaluation of two queries over the largest LUBM dataset in HBase took over 1s: $Q8$ (10s) and $Q9$ (1.5s). In contrast, six LUBM

Table III
LUBM QUERY COMPLEXITY AND EVALUATION PATTERNS.

Query complexity (# of triple patterns)	LUBM queries and their evaluation patterns
1	$Q6(C)$, $Q14(B)$
2	$Q1(A)$, $Q3(A)$, $Q5(C)$, $Q10(D)$, $Q11(A)$, $Q13(D)$
3	N/A
4	$Q7(A)$, $Q12(E)$
5	$Q4(C)$, $Q8(C)$
6	$Q2(B)$, $Q9(D)$

queries took over 1s in MySQL Cluster under similar circumstances. Finally, $Q1$, $Q3$, $Q7$, and $Q11$ of LUBM scaled significantly worse in MySQL Cluster.

E. Summary

Our performance study revealed interesting patterns in query evaluation, showed that our algorithms are efficient, and suggested that cloud computing has a great potential for scalable Semantic Web data management. Given that the experiments were performed with large datasets on commodity machines, both HBase and MySQL Cluster approaches showed to be quite efficient and promising. The proposed approaches were up to the task of efficiently storing and querying large RDF datasets. Overall, the experimental results were in favor of the HBase approach: not only were larger datasets able to load, but query performance and scalability were shown to be superior in many cases.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we studied the problem of distributed Semantic Web data management using state of the art cloud and relational database technologies represented by HBase and MySQL Cluster. We designed a novel database schema for HBase to efficiently store RDF data and proposed scalable querying algorithms to evaluate SPARQL queries in HBase. We chose a generic RDF database schema for MySQL Cluster and presented a SPARQL-to-SQL translation algorithm that generates flat SQL queries for SPARQL basic graph patterns. Finally, we conducted an experimental comparison of the two proposed approaches on a cluster of commodity machines using datasets and queries of the Third Provenance Challenge and Lehigh University Benchmark. Our study concluded that, while both approaches were up to the task of efficiently storing and querying large RDF datasets, the HBase solution was capable of dealing with larger RDF datasets and showed superior query performance and scalability. We believe that cloud computing has a great potential for scalable Semantic Web data management.

In the future, we will focus on architectural aspects of an RDF database management system in the cloud, search for optimizations in schema design, explore additional SPARQL features, and research inference support in distributed environments.

REFERENCES

[1] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems." *Journal of Web Semantics*, vol. 3, no. 2-3, pp. 158–182, 2005.

[2] A. Chebotko and S. Lu, *Querying the Semantic Web: An Efficient Approach Using Relational Databases*. LAP Lambert Academic Publishing, 2009.

[3] M. F. Husain, L. Khan, M. Kantarcioglu, and B. M. Thuraisingham, "Data intensive query processing for large RDF graphs using cloud computing tools," in *Proc. of CLOUD*, 2010, pp. 1 – 10.

[4] J. Abraham, P. Brazier, A. Chebotko, J. Navarro, and A. Piazza, "Distributed storage and querying techniques for a Semantic Web of scientific workflow provenance," in *Proc. of SCC*, 2010, 178-185.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, 2008.

[6] J. Myung, J. Yeon, and S. Lee, "SPARQL basic graph pattern processing with iterative MapReduce," in *Proc. of MDAC*, 2010, pp. 6:1–6:6.

[7] P. Ravindra, V. V. Deshpande, and K. Anyanwu, "Towards scalable RDF graph analytics on MapReduce," in *Proc. of MDAC*, 2010, pp. 5:1–5:6.

[8] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen, "Scalable distributed reasoning using MapReduce," in *Proc. of ISWC*, 2009, pp. 634–649.

[9] A. Matono, S. M. Pahlevi, and I. Kojima, "RDFCube: A P2P-based three-dimensional index for structural joins on distributed triple stores," in *Proc. of DBISP2P Workshops*, 2006, pp. 323–330.

[10] M. Cai, M. R. Frank, B. Yan, and R. M. MacGregor, "A subscribable peer-to-peer RDF repository for distributed metadata management," *Journal of Web Semantics*, vol. 2, no. 2, pp. 109–130, 2004.

[11] B. Quilitz and U. Leser, "Querying distributed RDF data sources with SPARQL," in *Proc. of ESWC*, 2008, pp. 524–538.

[12] H. Stuckenschmidt, R. Vdovjak, J. Broekstra, and G.-J. Houben, "Towards distributed processing of RDF path queries," *International Journal of Web Engineering and Technology*, vol. 2, no. 2/3, pp. 207–230, 2005.

[13] N. Li, J. Rao, E. J. Shekita, and S. Tata, "Leveraging a scalable row store to build a distributed text index," in *Proc. of CloudDb*, 2009, pp. 29–36.

[14] H. Choi, J. Son, Y. Cho, M. K. Sung, and Y. D. Chung, "SPIDER: a system for scalable, parallel/distributed evaluation of large-scale RDF data," in *Proc. of CIKM*, 2009, pp. 2087–2088.

[15] A. Chebotko, S. Lu, and F. Fotouhi, "Semantics preserving SPARQL-to-SQL translation," *Data & Knowledge Engineering*, vol. 68, no. 10, pp. 973–1000, 2009.