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

TITLE: Research and Design of RDF Storage System based on HBase

Author: JIN, Qiang

Supervisor: SUN, Jianling

Subject: Computer Application

College: Computer Science

Submitted Date: Jan, 2011

摘要

随着语义 Web 的发展，资源描述框架（RDF）得到了广泛的应用。然而传统集中式的 RDF 存储系统在日益增长的数据面前遭遇了难以跨越的存储与查询瓶颈。研究人员开始将目光投向分布式领域，以期利用分布式系统所具备的海量存储与并行计算能力来解决当前集中式 RDF 存储系统面临的各项问题。

本文以 RDF 存储系统为研究对象，提出了采用分布式存储系统 HBase 存储 RDF 数据的方案，以及应用 MapReduce 并行计算框架进行 RDF 查询处理的策略。

首先，本文介绍了 RDF 存储系统的研究现状。本文介绍了 RDF 的概念背景、RDF 标准查询语言 SPARQL 的构成，简述了当前已有的部分分布式 RDF 存储系统实现以及当前基于 Hadoop 相关技术的 RDF 存储研究现状。

接着，在深入分析 RDF 存储系统的各项特性后，本文提出采用分布式存储系统 HBase 来存储 RDF 数据的具体方案。RDF 数据将被存储在 SPO、POS、OSP 三张表中。本文描述的方案充分利用了 HBase 的默认索引机制，在保证 RDF 查询性能的同时有效地减少了 RDF 数据的存储开销。

然后，本文提出采用 MapReduce 并行计算框架处理 SPARQL 查询语言中 Basic Graph Pattern（BGP）的具体策略。本文在当前已有的 MapReduce 多路连接方法之上提出了一个贪心的多路连接选择策略。本文提出在进行 MapReduce 多路连接时优先处理具备高选择性的 Triple Pattern 子句，这样就可以在 Map 阶段提前过滤冗余数据，从而在保证 MapReduce 任务数目一定的情况下尽可能减少整个连接处理过程中的 I/O 操作开销。本文采用 LUBM 测试集对查询策略进行了实验，实验结果表明了本文提出的查询策略在大数据集下可以有效工作。

最后，本文以 HBase 上的 RDF 存储方案与 MapReduce 连接处理策略为基础搭建了基于 HBase 的 RDF 存储系统原型。

关键词： 资源描述框架，分布式系统，HBase，MapReduce 并行计算框架

Abstract

Since the rapid development of semantic web technologies, Resource Description Framework is widely used nowadays. However, the traditional centralized RDF stores have limitations in handling huge RDF datasets. To resolve the problem, distributed and parallel system are now be introducing into RDF storage system.

In this paper, we researched on RDF storage system and proposed using HBase, which is a distributed column-oriented database, to store RDF datasets and using MapReduce to answer RDF queries.

First, we introduced the background knowledge of modern RDF storage system, including the concept of RDF and standard RDF query language SPARQL. We then gave an overview of existing distributed RDF storage system and the current researches on integrating RDF store with Hadoop related technologies.

Then, with a deep analytical understanding of RDF storage system, we proposed an approach to use HBase to store RDF dataset. RDF triples will be stored in three HBase tables, which are SPO、POS and OSP. Our approach makes full use of the default index structure provided by HBase, which promised the respond time for query with reduced storage space.

After that, we proposed a MapReduce strategy for handing SPARQL Basic Graph Pattern (BGP). We suggested that high selecting triple patterns and small intermediate results should be included in MapReduce job first. We proposed a greedy query plan generating strategy based on existing MapReduce multi-way joins researches. The evaluation result shows that our approach works well against large RDF dataset.

Finally, we built a demo RDF storage system based on the proposed HBase RDF schema and MapReduce query answer strategy.

Keywords: RDF, Distributed system, HBase, MapReduce

目录

图目录

表目录

第1章 绪论

1.1 课题背景

互联网经过多年的发展，已逐步成为全球最大的信息仓库。人们可利用的信息空前丰富，但如何查找有效的信息却成为了一个日益困难的问题。当前的互联网可看作是一个文档构成的网络，其上的信息无法被机器自动理解处理。这样就不可避免陷入了“数据丰富，信息贫乏（Rich Data，Poor Information）”的境地[1]。

Tim Berners-Lee 提出的语义 Web 正是为了解决上述问题。语义 Web 的目的在于让分布网络各处的信息具有一定语义，从而能够被计算机理解处理，进而将人从寻找信息的桎梏中解放出来[2]。

语义 Web 的层次结构总共经历过四个版本的变更，其层次结构如图 1.1 所示。虽然语义 Web 的层次架构不断被研究人员们修正，但各个版本的架构中 Resource Description Framework（RDF）[3]作为语义 Web 的基石这一点不曾改变。

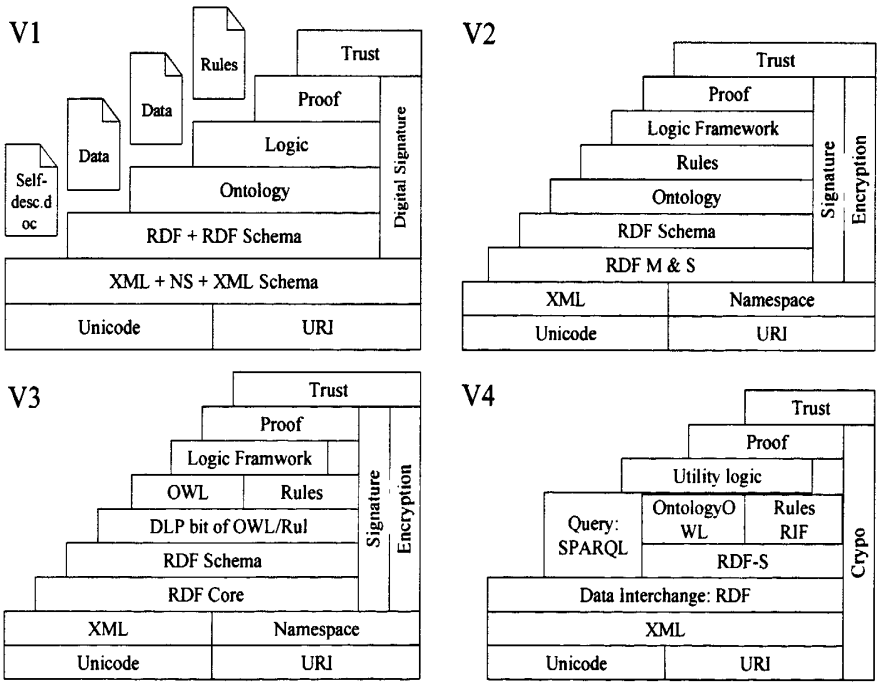


图 1.1 语义 Web 层次架构的四个版本

RDF 是 W3C 提出的用于描述网络资源的标准。它的设计理念是希望在最低限度的约束之上更为灵活的描述资源信息，并且希望其数据模型能够独立于不同的应用，同时又便于在不同应用间进行数据整合。故而 RDF 采用了一种简易的描述方式，即用主体（Subject），谓词（Predicate），客体（Object）构成的三元组来表示资源。RDF 数据通用表示形式为（S，P，O）三元组，一组 RDF 数据可构成一个 RDF 有向图。

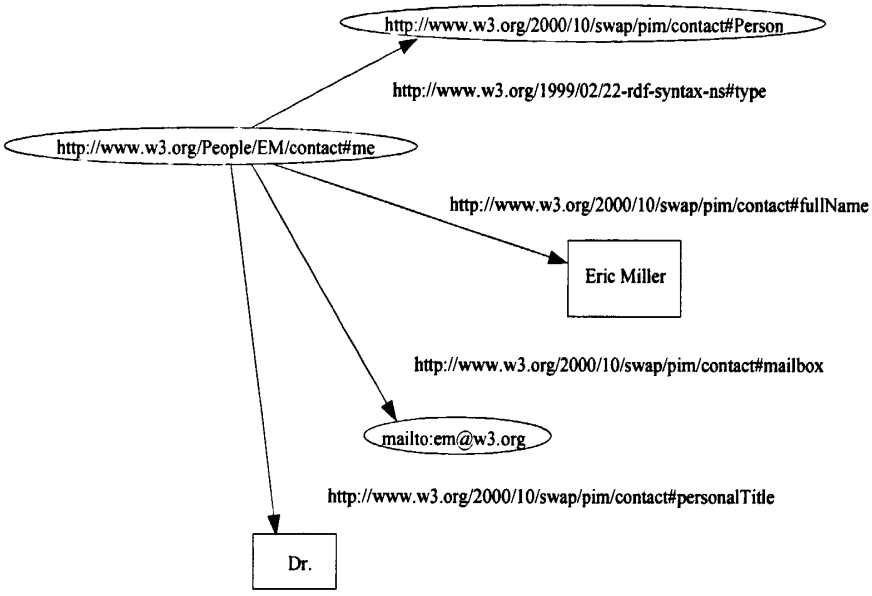


图 1.2 RDF 有向图

RDF 是一种半结构化的数据格式，但其（S，P，O）三元组的形式却又十分易于存储。早期 RDF 存储系统采用单一机器集中化的架构模型，通过关系数据库来存储查询 RDF 数据。但是随着语义 Web 技术的发展，RDF 数据格式被越来越多的采用在各种系统当中，如互联网应用、生物技术系统等等。随着计算机技术进步而来的是各种系统所处理的数据量也与日俱增。实际应用中的 RDF 数据集的三元组数目同早期相比已不可同日而语，譬如当前研究领域的一个热点就是设计实现能够高效处理以十亿为单位的超大 RDF 数据集。在这样爆炸式增长的存储压力之下，原始的简易架构模型无可避免的遭遇了性能瓶颈，在存储能力到查询响应性能方面都不能满足日益增长的需求。

在巨大的数据压力面前，研究人员对 RDF 存储系统的各个方面如 RDF 存储模型定义、底层存储系统选则、查询连接处理等均进行了详细的分析研究，从不同角度提出了新的解决方案。在 W3C RDF 官方网页上就可发现有相当多的 RDF 存储系统方案，与此同时研究人员也继续在不断拓展新的设计思路。但当前依然没有一种被业界完全认可的 RDF 存储系统方案。不过从多项研究的进展来看，将 RDF 存储系统同分布式系统相结合是一个大的趋势。

分布式系统研究领域近年来得到了很大的发展。Google 发表的分布式文件系统 Google File System（GFS）[4]、分布式数据库 Bigtable[5]与并行计算框架 MapReduce[6]等成果是当前分布式系统研究领域的热点之一。上述技术在 Google 的各项应用中得到了大量的使用，在开源社区出现了与之相对应的开源实现。Apache Hadoop[7]项目包含了分布式文件系统 Hadoop Distributed File System（HDFS）、并行计算框架 MapReduce，而 Apache HBase[8]则是 Bigtable 的开源实现。这些开源分布式项目的涌现丰富了研究人员的选择。当前已经有不少研究人员试图将 Hadoop 相关的各项技术与 RDF 相整合。然而在这方面的研究还处在一个初始阶段，当前还未出现一个成熟的系统方案。

1.2 本文的研究内容

本文主要分析了 RDF 存储系统的研究现状，认定将 RDF 存储系统与分布式系统相结合的思路。然而创建一个可靠健壮完善高效的分布式系统并不是一项简单的工作，因此我们觉得最适宜的方式是整合现有的分布式框架，尽可能利用已有的一切资源才能帮助我们快速的搭建一个可用在生产环境的 RDF 存储系统。因此本文提出了一个基于 HBase 的 RDF 存储系统的设计方案。本文的主要工作如下：

1) 分析了 RDF 存储系统现状。本文分析了当前几种经典的 RDF 数据存储模型，随后介绍了当前已有的部分分布式 RDF 存储系统方案以及当前基于 Hadoop 相关技术的 RDF 存储查询研究现状。
2) 提出了 HBase 之上的 RDF 数据存储模型。本文在介绍了 HBase 的架构特

性后分析了几种经典 RDF 数据模型应用在 HBase 上的方案与优缺点。在比较了各种方案的优劣后，本文提出了一种基于 Hexastore[9]的 RDF 数据存储方案。在本文的方案中，RDF 数据将存放在如下三张 HBase 数据表当中：SPO、POS 与 OSP。本文描述的方案充分利用了 HBase 的默认索引机制，在保证 RDF 查询性能的同时有效地减少了 RDF 数据的存储开销。

3) 提出了基于 MapReduce 的 RDF 查询处理策略。本文在已有的 MapReduce 多路连接方案之上提出了一个贪心的多路连接选择策略。本文提出在进行 MapReduce 多路连接时优先处理具备高选择性的 Triple Pattern 子句，这样就可以在 Map 阶段提前过滤冗余数据，从而在保证 MapReduce 任务数目一定的情况下尽可能减少整个连接处理过程中的 I/O 操作开销。本文采用 LUBM[10]测试集对查询策略进行了实验，实验结果表明了本文提出的查询策略在大数据集下可以有效工作。

4) 搭建了基于 HBase 的 RDF 存储系统原型。本文构建了基于 HBase 的 RDF 存储系统原型，原型系统支持 RDF 数据存储与查询。原型系统部署在 HBase 集群之上，系统的可靠性通过底层 HBase 来提供，系统的计算能力通过 MapReduce 并行计算框架来保证。

1.3 本文的组织结构

根据上文介绍的研究内容，本文剩余部分章节组织如下：

第 2 章：RDF 存储系统现状。这一章分为四部分：第一部分对 RDF 标准查询语言 SPARQL[11]进行了简要介绍；第二部分对当前经典的 RDF 存储模型进行了分析；第三部分介绍了当前已有的部分分布式 RDF 存储系统；第四部分概述了当前基于 Hadoop 相关技术的 RDF 存储研究现状。

第 3 章：基于 HBase 的 RDF 存储系统原型设计。这一章分为三部分：第一部分提出了原型系统的设计目标；第二部分对原型系统设计的关键技术 HBase、ARQ[12]、MapReduce 进行了介绍；第三部分介绍了原型系统的整体架构与模块。

第 4 章：RDF 存储模型定义。这一章分为三个部分：第一部分详细分析了

HBase上经典RDF存储模型的应用方案及优劣；第二部分分析介绍了本文提出的RDF存储模型。第三部分介绍了RDF数据导入HBase的流程。

第5章：RDF查询处理。这一章分为两个部分：第一部分介绍了SPARQL查询预处理过程；第二部分详细介绍了MapReduce连接处理策略。

第6章：测试与实验。这一章对本文提出的RDF查询处理策略进行了实验分析。

第7章：总结与展望。这一章回顾了本文的主要内容并对研究结果进行了总结，此外也进一步指出了当前研究的问题与缺陷及今后研究改进的方向。

1.4 本章小结

本章首先介绍了RDF相关的背景知识，并阐述了传统RDF存储系统在海量数据前面临的压力与挑战。随后本章提出了本文的具体研究内容，即采用分布式解决方案HBase来构建高可扩展性的RDF存储系统。最后本章概述了本文后续章节的内容安排。

第2章 RDF 存储系统研究现状

RDF 存储系统的核心在于 RDF 数据的存储组织与 RDF 查询语言的处理。本章将首先对 RDF 标准查询语言 SPARQL 以及经典的 RDF 数据存储方案（Triple Store、Property Table、Vertical Partitioning、Hexastore）进行分析。在这之后，本章将对当前部分分布式 RDF 存储系统（RDFPeers、YARS2、4store、Clustered TDB）进行简要介绍。本章的最后将会介绍分析当前基于 Hadoop 的 RDF 存储查询研究现状。

2.1 SPARQL 概述

SPARQL 是 W3C 提出的 RDF 标准查询语言，它的语法同关系数据库查询语言 SQL 接近。图 2.1 展示了一个 SPARQL 语句，该查询语句包含了一个由六个 Triple Pattern 子句组成 Basic Graph Pattern（BGP）。SPARQL 语言的特点在于其查询构建在模式匹配上。Triple Pattern 是 SPARQL 中最基本的匹配单元，多个 Triple Pattern 子句可以组成更为复杂的模式匹配块，如 BGP 等。常见的 SPARQL 模式匹配方式除了 BGP 之外还有如下几种：Group Graph Pattern、Optional Graph Pattern、Alternative Graph Pattern、Patterns on Named Graph。

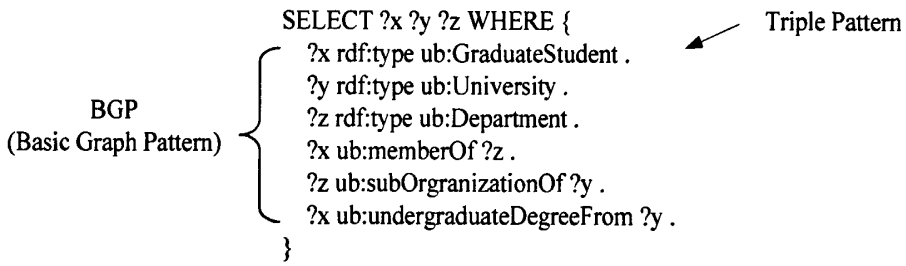


图 2.1 SPARQL 示例

Triple Pattern 作为 SPARQL 中最基本的匹配单元，其构成与 RDF 三元组表示形式相对应。RDF 数据以（S，P，O）三元组表示，Triple Pattern 也由这三部分构成。Triple Pattern 对应位置可以是绑定了的值或是未绑定的变量。未绑定变量由问号加变量名组成。多个 Triple Pattern 子句间共用同一个变量则表示这些子

句间存在连接关系。Triple Pattern 子句的表现形式及语义如表 2.1 所示。

表 2.1 Triple Pattern 定义

	表达式	语义
Q1	(S P O)	若存在则返回该三元组否则返回空
Q2	(S P ?O)	给定 Subject、Predicate，返回对应三元组中 Object 值
Q3	(S ?P O)	给定 Subject、Object，返回对应三元组中 Predicate 值
Q4	(S ?P ?O)	给定 Subject，返回对应三元组中 Predicate 与 Object 值
Q5	(?S P O)	给定 Predicate、Object，返回对应三元组中 Subject 值
Q6	(?S P ?O)	给定 Predicate，返回对应三元组中 Subject 与 Object 值
Q7	(?S ?P O)	给定 Object，返回对应三元组中 Subject 与 Predicate 值
Q8	(?S ?P ?O)	返回所有的三元组

2.2 RDF 存储模型概述

RDF 基本结构十分简洁即 RDF 三元组（Subject, Predicate, Object）。RDF 数据存储系统当前主要依托 RDBMS（关系型数据库），此外也有部分系统采用其余一些底层存储方案。不论 RDF 数据最终存储在何种系统当中，RDF 存储系统都必须首先关注如何存储组织 RDF 三元组。这一小节将介绍当前流行的集中 RDF 存储结构模型。

2.2.1 Triple Store

在采用关系数据库存储 RDF 数据时，Triple Store 是最为直观的一种 RDF 数据组织方式。在 Triple Store 中，所有的 RDF 三元组数据存储在一张数据表当中。该数据表由三列构成，分别对应 Subject、Predicate、Object。其结构如表 2.2 所示。

表 2.2 Triple Store 示例

Subject	Predicate	Object
ID1	Type	Computer
ID1	Brand	Lenovo

Triple Store 的优点如下：

- 结构简洁。Triple Store 的表结构完全对应 RDF 三元组形式，十分易于搭建在关系数据库之上。

Triple Store 的缺点如下：

- 自连接（Self-Join）过多。Triple Store 中每一个数据行对应一个 RDF 三元组，同一个 Subject 下不同属性的三元组存放在不同的行当中。因此在响应 RDF 查询时往往会生成大量的自连接操作。
- 数据表过大。Triple Store 只采用一张数据表来存储 RDF 数据，每一行对应一个 RDF 三元组。当前某些 RDF 数据集往往含有千万级甚至是更多的 RDF 三元组，这种情况下单张数据表进行存储就难以满足查询响应性能。

2.2.2 Property Table

Property Table 的提出主要是为了避免 Triple Store 中存在的过多自连接操作。Jena Semantic Web Toolkit 2[13]中引入了 Property Table 作为框架中的一种 RDF 存储组织形式。Property Table 有两种不同类型：Clustered Property Table、Property Class Table。

Clustered Property Table 方案中引入了 Cluster 聚集算法用以将多个可能同时出现在一个 Subject 中的属性聚集存放在一张表中。其结构定义如表 2.3 所示，数据表的主键是 Subject，数据表的列数以及名称由 Cluster 算法确定。在 Clustered Property Table 中，同一个 Subject 的多个属性可能分别出现在几张不同的数据表当中，但一种类型的属性最多只能出现在一张数据表中。

表 2.3 Clustered Property Table 示例

Subject	Type	Price	Brand
ID1	Car	200，000	Benz
ID2	Computer	8，000	Lenovo
ID3	Computer	7，000	Dell

Property Class Table 方案中同种类型的 Subject 将会被存放在同一张表中。其结构定义如表 2.4 所示，数据表的主键是 Subject，数据表的列对应该 Subject 拥有的属性。在 Property Class Table 中，一个属性可以出现在多个数据表中，不同数据表中的同名列分属不同的 Subject，彼此间没有冲突。

表 2.4 Property Class Table 示例

Type：Computer

Subject	Price	Brand
ID2	8，000	Lenovo
ID3	7，000	Dell
ID4	7，500	HP

Type：Car

Subject	Price	Brand
ID5	200，000	Beetle
ID6	500，000	BMW
ID7	1，000，000	Benz

Property Table 的优点如下：

- 自连接操作减少。Property Table 的一个数据行中存放了同一个 Subject 下的多个属性，因此在响应查询响应的时候能够一定程度上减少自连接操作。如若 RDF 查询的属性均存放在一个数据行当中，则可以完全避免自连接操作。

Property Table 的缺点如下：

- 实现复杂。Property Table 通常底层也采用关系数据库进行存储，Property Table 需要引入 Cluster 算法来将不同的属性进行聚集，又需控制属性集合的数目以免在一张数据表中生成过多的列。数据表结构与 Triple Store 相比负责很多。
- 存在空值（NULL）。Property Table 在对不同属性进行聚集时，由于 RDF 数据本身本结构化的稀疏特性，在数据库中进行存储时不可避免的出现

大量空值。

- 存在 Property Table 间的连接操作。真实应用场景中的 RDF 查询通常需要同时读取大量属性，这些属性可能分属不同的数据表当中。这样就需要对不同数据表进行连接操作。Property Table 虽然一定程度上减少了自连接操作，但却引入了多个数据表之间的连接操作。
- 对 Multi-Valued 属性支持不好。Multi-Valued 属性是指一个 Subject 的 Predicate 对应了多个值，比如一个人的地址可以有多份，用 RDF 进行表示时地址在这时就是一个 Multi-Valued 属性。RDF 数据中通常存在大量 Multi-Valued 属性，Property Table 不能很好的解决 Multi-Valued 属性的存储问题。

2.2.3 Vertical Partitioning

在 Vertical Partitioning[14]方案中，RDF 数据集中的每一个属性都有一个对应的数据表。该表中存放了所有拥有该属性的 RDF 三元组数据。数据表本身已经隐含了属性值，因此数据表中只存在两列，分别对应 RDF 三元组中的 Subject、Object。Triple Store 只包含一张数据表，Property Table 则根据不同属性的聚集情况创建多张数据表，Vertical Partitioning 却是为每一个属性都创建了一张数据表。其结构定义如表 2.5 所示。

表 2.5 Vertical Partitioning 示例

Predicate: Price

Subject	Object
ID2	8，000
ID3	7，000

Predicate: Brand

Subject	Object
ID2	Lenovo
ID3	Dell

Vertical Partitioning 的优点如下：

- 结构简单。Vertical Partitioning 针对每个属性都创建了一张表进行存储，每张数据表均只包含对应 Subject、Object 的两列。对比 Property Table 中的复杂表定义，其结构简单许多。
- 避免空值。Vertical Partitioning 中每个数据表只存储有值的三元组，避免了 Property Table 中存在的空值问题。
- 支持 Multi-Valued 属性。Vertical Partitioning 并不区分属性是 Multi-Valued 或是 Single-Valued，Multi-Valued 属性值存储在不同的数据行当中，彼此间无任何关联。

Vertical Partitioning 的缺点如下：

- 数据表过多。Vertical Partitioning 需要为每一个属性都创建一张表进行存储，然而真实的 RDF 数据集可能包含大量不同的属性，这种情况下就会导致系统产生过多的表。在采用关系数据库进行存储时，首先数据表的数目受限于数据系统的支持，其次数据表过多也会导致维护困难。
- 表间连接过多。RDF 查询可能需要读取大量不同属性，而在 Vertical Partitioning 方案中，不同属性数据存储在不同的表当中。因此在响应查询时不得不进行大量的表之间的连接操作。
- 不支持属性未知的查询。Triple Pattern 子句中的属性值可能未绑定，此时由于无法获取到属性名称，RDF 存储系统需要遍历所有的数据表才能完成查询响应，这在性能上是难以接受的。

2.2.4 Hexastore

不同于 Triple Store、Property Table 以及 Vertical Partitioning 通过数据表来存储 RDF 数据，Hexastore 是一种 RDF 数据索引方式，RDF 数据存储在 Hexastore 描述的多个索引当中。

针对 Triple Pattern 可能的表现形式，Hexastore 提出对 RDF 三元组创建六个不同索引结构来分别应答不同的 Triple Pattern 查询。Hexastore 方案中包含了如下六个索引：SPO、SOP、PSO、POS、OSP、OPS。这六个索引的结构类似，图 2.2

展示了 SPO 索引项的大体结构。SPO 索引首先通过 Subject 来查询索引项，每一个索引项中存在一个链表，链表中的元素对应该 Subject 拥有的属性。链表的每一个元素同时也拥有一个指针指向一个由 Object 构成的链表。属性对应的值存放在该 Object 链表当中。SPO 索引需要链表中的各项元素保证有序，这样能提供更高效的查询。

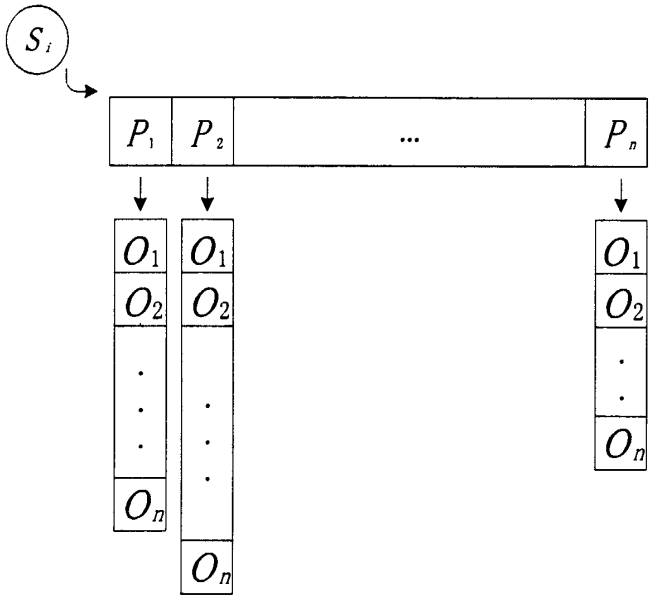


图 2.2 Hexastore 索引结构示例

Hexastore 的优点在于：

- 支持 Multi-Valued 属性。Multi-Valued 属性对应的多个值均可存放在 Hexastore 索引结构的 Object 链表当中。
- 避免空值。Hexastore 索引结构之存储有值的 RDF 三元组，因而可以避免 RDF 数据存放在关系数据库中的空值问题。
- 降低 I/O 开销。Hexastore 中的不同索引用于响应不同的 Triple Pattern 查询，这种查询应答方式避免了无用数据的访问，从而减低了 I/O 开销。
- 连接操作高效。Hexastore 的多个索引均需保证索引项中链表结构的有序行，这样在进行连接操作时可以采用高效的归并连接方式。

Hexastore 的缺点在于：

- 大量的索引存储空间。Hexastore 需要创建六个不同索引结构，每一个索引中都包含了完整的 RDF 数据。这就需要大量的存储空间。
- 索引创建过程复杂。RDF 数据导入时需要对六个索引结构进行更新，在更新的同时需保证各索引项中的有序性，这使得索引创建过程变得复杂。此外多个索引的创建也对 RDF 数据导入的性能产生疑问。
- 数据修改操作复杂。RDF 数据存储在六个索引结构之中，在对数据进行修改时需要同时更新六个索引，数据的修改操作会变得复杂低效。
- 索引结构实现困难。Hexastore 采用的索引结构不能很好的直接采用关系数据库或是其它存储系统来实现，开发人员需要自己编写代码来实现维护索引结构。此外随着 RDF 数据量的不断增长，单一机器将会难以管理多个巨大的索引。在分布式环境下创建维护 Hexastore 描述的索引结构并不是一件容易的事情。

2.3 分布式 RDF 存储系统概述

传统集中式的 RDF 存储系统无法适应爆炸式增长的海量 RDF 数据，因此研究人员将目光投向了分布式领域。分布式系统所具备的海量存储与并行计算能力使其被认为是解决海量 RDF 数据存储查询难题的一个适宜途径。分布式系统随着计算机技术的进步也有了长足的发展，当前在 RDF 领域已经出现了一些基于通用分布式技术的存储方案与一些为 RDF 数据存储专有优化过的集群方案。本小节将对部分已有的系统进行相应介绍。

2.3.1 RDFPeers

RDFPeers [16]是一个基于 P2P（Peer-to-Peer）技术的分布式 RDF 存储系统。它构建在 MAAN（Multi-Attribute Addressable Network）之上。RDFPeers 集群中的每个节点由如下几个模块组成：MAAN Network Layer、RDF Triple Loader、RDF Triple Storage、Native Query Reslover、RDQL Translator。在 RDF 数据存储上，一

个 RDF 三元组（S，P，O）按照 Subject、Predicate、Object 的哈希值被索引三次，并根据哈希值的不同划分到不同节点中。在 RDF 查询上，RDFPeers 根据查询选择响应节点，并通过节点上的 RDQL Translator 与 Native Query Resolver 进行处理。

2.3.2 YARS2

YARS2 [17]是一个为了 RDF 读取操作进行过大量优化的集群式 RDF 存储系统。在 RDF 数据存储方面，YARS2 采用了六个索引来存储 RDF Quad（Subject、Predicate、Object、Context）。在索引结构上，YARS2 实现了称为 Sparse Index 的索引结构。Sparse Index 是一个内存数据结构，其中的索引项指向了磁盘上每一个 RDF 数据文件的起始位置。在进行查询时，YARS2 在内存中对 Sparse Index 进行二分查找，从而定位所需的 RDF 数据文件所在。YARS2 中的 RDF 数据文件采用 Huffman 编码进行压缩。在数据划分上，YARS2 也采用了哈希的方式。

2.3.3 4store

4store[18]是另一个集群式 RDF 存储系统。在 RDF 数据存储上，4store 支持存放 RDF Quad（Model、Subject、Predicate、Object）。在数据划分上，4store 为每个 Subject 分配一个 RID，而后根据 RID 将 RDF 数据划分给不同的 Segment 并存放在不同节点当中。节点中的 RDF 数据存放在三个不同索引当中：P Index、M Index、R Index。P Index 用于索引 Predicate，由 Radix Trie 构成。M Index 用于索引 Model，其实现为哈希表。

2.3.4 Clustered TDB

Clustered TDB [19]是 Jena 的集群版本。在 RDF 数据存储上，Clustered TDB 中不存在 Triple Table 似的表结构，RDF 三元组是被存储与 SPO、POS、OSP 三个 B+树索引结构当中。在数据划分上，Cluster TDB 对 Subject、Predicate、Object 分别应用哈希函数，根据哈希值来决定数据划分到哪个存储节点当中。在 RDF 查询上，Query Coordinator 负责判定哪些节点可以用于查询响应。

2.4 基于 Hadoop 的 RDF 存储研究概述

Hadoop 项目包含了 Google File System 分布式文件系统、MapReduce 并行计算框架的开源实现。随着 Hadoop 被广泛应用，它的存储与计算能力得到了认可。在 RDF 研究领域，已经有不少研究人员试图将 Hadoop 与 RDF 相结合，以期利用 Hadoop 提供的海量存储与并行计算能力来解决超大规模 RDF 数据的存储查询问题。在这方面的研究现在还处在起步阶段，目前的大部分工作集中在 RDF 数据在 Hadoop 上的存储设计以及 SPARQL BGP 连接操作的 MapReduce 并行策略上。

M.F. Husain 等人提出了一个在 Hadoop 上存储 RDF 数据并采用 MapReduce 处理 RDF 查询的方案 [20]。在 RDF 数据存储方面，他们提出将 RDF 数据以文件形式存放在 HDFS 分布式文件系统当中。RDF 数据首先需进行两阶段预处理操作（Predicate Split、Predicate Object Split）来缩减 RDF 数据大小。在 RDF 查询处理方面，他们提出了一个贪心的 MapReduce 任务生成算法。多个 MapReduce 任务迭代处理 SPARQL BGP 连接操作，每个 MapReduce 任务优先处理共享变量出现次数最多的 Triple Pattern 子句。上述方案存在如下一些缺陷：RDF 数据首先需经多步预处理操作；RDF 数据直接存放在 HDFS 上，缺少高效的索引结构；MapReduce 查询响应策略过于简单并不能有效保证查询响应效率。

Jaeseok Myung、Jongheum Yeon 与 Sang-goo Lee 给出了一个 MapReduce 多路连接策略以处理 RDF 标准查询 SPARQL 中的连接操作 [21]。在 RDF 数据存储方面，他们也是将 RDF 数据以 N-Triples 文件格式直接存放在 HDFS 当中。在 RDF 数据查询方面，他们的方法首先需要创建一个 MapReduce 任务从 HDFS 中的 RDF 数据文件中查找对应的 RDF 三元组，随后再创建多个 MapReduce 任务迭代处理 SPARQL BGP 连接操作。在 MapReduce 连接处理方面，他们给出了一个贪心的连接选择策略与一个多路连接选择策略。通过定义 MapReduce 任务的输入输出格式，使得一个 MapReduce 任务可以同时处理两个不同的连接操作。上述方案的主要缺陷在于未对 RDF 数据创建有效的索引结构，在处理查询时首先需要创建一个 MapReduce 任务遍历所有的 RDF 数据来选取每个 Triple pattern 子句对应的结果。

此外他们没有给出在贪心连接策略与多路连接策略之间的选取方法。

HadoopRDF [22]是 Yuan Tian 等人提出的一个 Hadoop 与传统 RDF 存储 sesame[23]相结合的 RDF 数据存储分析系统。HadoopRDF 的构想是在 Hadoop 集群的每一个节点上部署一个 sesame server 实例，通过 sesame 的接口来提供 RDF 数据存储查询服务。系统之所以引入 Hadoop 是为了利用 Hadoop 集群的高可靠性与高容错恢复能力。论文中简述了系统的整体架构，但却并未透露 RDF 数据存储与查询处理的细节实现，因此难以评估系统的优劣。

Hyunsik Choi 等人在一篇 demo 论文中描述了一个基于 HBase 的 RDF 存储系统 [24]。论文只简述了系统中 Graph Loader、Graph Query Processor 的功能作用，并未给出其实现方案。此外论文里也没有提及 RDF 数据在 HBase 上的存储组织方式。

2.5 本章小结

本章详细介绍了 RDF 存储系统的研究现状。本章首先介绍了当前 RDF 标准查询语言 SPARQL，随后分析了经典的 RDF 存储模型 Triple Store、Property Table、Vertical Partitioning、Hexastore 等的优劣。在了解了 RDF 存储相关的基本信息后，本章分析介绍了当前已有的部分分布式 RDF 存储系统：RDFPeers、YARS2、4 store、Clustered TDB。本章最后部分介绍了与当前热门的分布式系统 Hadoop 相关的 RDF 存储研究进展。

第3章 基于HBase的RDF存储系统原型设计

本章将介绍分析基于 HBase 的 RDF 存储系统的整体设计方案。首先本章会明确原型系统的设计目标，随后本章对原型系统涉及的关键技术 HBase、ARQ、MapReduce 进行必要介绍，最后本章会给出原型系统的架构及功能模块设计。

3.1 设计目标

这一小节首先明确本文提出的基于 HBase 的 RDF 存储系统的设计目标，对于当前的原型系统来说，主要的设计目标包括：

- 支持 SPARQL 查询语言中的 Triple Pattern 查询以及 BGP 连接操作。对于 Triple Pattern 以及简单的 BGP 连接，原型系统需要保证查询效率。
- 平滑应对 RDF 数据增长。原型系统需要保证存储查询功能的可伸缩性，系统应当能够通过硬件资源的扩充来应对 RDF 数据不断增长的挑战。

3.2 技术路线

为了实现上述设计目标，本文选取了如下关键技术作为系统搭建的基础设施。在数据存储上，本文选用了 HBase 分布式数据库；在 SPARQL 查询解析上，本文选用了 ARQ 解析工具；在 BGP 连接处理上，本文选用了 MapReduce 并行计算框架。本节将详细介绍上述各项技术。

3.2.1 HBase

HBase 是构建在 Apache Hadoop 之上的分布式数据库。HBase 可以看作是 Google Bigtable 的开源版本，实现了 Google Bigtable 提出的各项特性。HBase 作为当前流行的 NoSQL 数据库的一种，已经得到了广泛的应用。

3.2.1.1 HBase 架构模型

HBase 分布式集群采用了经典的 Master-Slave 模型。HBase 集群节点分为：HMaster、HRegionServer。HMaster 是集群主节点，主要负责维护 HBase 集群内

的各种元数据信息。HRegionServer 则负责具体的 HBase 数据读写任务。其整体架构如图 3.1 所示。

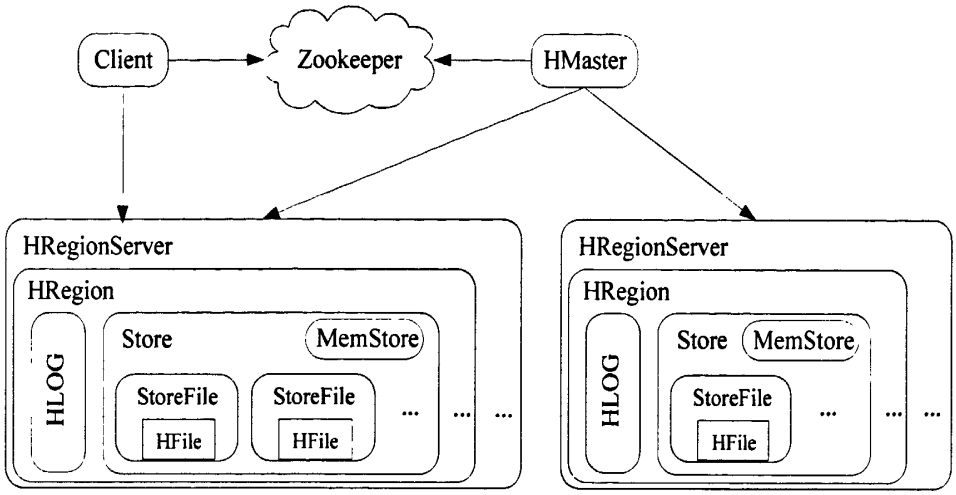


图 3.1 HBase 存储架构模型

HBase 可以支持千万行的数据表，当 HBase 数据表内数据容量达到一定容量阀值时，其底层数据文件将会分裂成多个 Region。一张数据表可能由数百个不同 Region 的数据构成。如图 3.1 所示，每一个 HRegionServer 均会管理多个 Region 数据，Region 同 HBase 数据表的匹配关系则由 HMaster 负责维护。

3.2.1.2 HBase 数据模型

HBase 提供了与关系数据库概念相近的数据表 HTable，但不同于关系数据库基于行进行存储，HTable 是基于列进行存储。HTable 的构成如图 3.2 所示。

Row-Key: Key1
Column Family: A
Column: X
T1 Value1
T2 Value2
Column: Y
T3 Value3
Column Family: B
Column: Z
T4 Value4

图 3.2 HTable 结构

HTable中的行由Row-Key以及一组Column Family构成。Row-Key是每一个数据行的唯一标识，Column Family用于存储数据。Column Family中可以动态添加任意数目的Column，理论上HBase不限制Column的数目。HBase的Column可以存放基于时间戳的多个版本数据。HTable中数据修改相当于在新的时间戳下生成新版本数据。基于HTable的构成，本质上来看HTable更接近于一张多维有序哈希表而非传统意义上数据库表。在实际应用中，HTable往往就扮演了分布式哈希表的作用，（Row-Key, Column Family : Column, Timestamp）-> Value。

Row-Key是HTable数据行的唯一标识，HBase默认对其提供了类似B+树的高效索引结构。在物理存储上HBase将Row-Key连续的数据划分到不同的Region当中，每个Region交由HRegionServer管理。

3.2.2 ARQ

ARQ是Jena框架中的SPARQL查询解析引擎，提供了对RDF标准查询语言SPARQL的完整支持。每个SPARQL语句通过ARQ解析后可以获取到SPARQL algebra表达式。

```
SELECT ?x WHERE {
 ?x rdf:type ub:GraduateStudent .
 ?x ub:takesCourse GraduateCourse0
}
```

(a) LUBM Q1

```
(project (?x)
 (bgp
  (triple ?x rdf:type ub:GraduateStudent)
  (triple ?x ub:takesCourse GraduateCourse0)
 )
)
```

(b) ARQ解析结果

图 3.3 SPARQL algebra生成

3.2.3 MapReduce

MapReduce概念起源于函数式编程语言中的Map与Reduce函数。Google在分布式集群环境上提出了一种接近函数式编程语言Map、Reduce概念的并行计算框架并以此命名。

3.2.3.1 MapReduce概念

MapReduce并行计算框架将任务处理划分为Map阶段与Reduce阶段。对于

分布式程序开发人员来说，应用 MapReduce 框架会极大简化分布式程序的设计。MapReduce 框架理论上只需要程序人员提供 Map 函数与 Reduce 函数的实现，框架负责了具体计算任务在集群中的分配以及数据在 Map 阶段与 Reduce 阶段的传输等繁杂细节。在 MapReduce 框架的帮助下，开发人员能专注于计算任务本身。

MapReduce 框架中 Map 与 Reduce 函数的输入都是一些列的（Key，Value）值，因此也可以看出 MapReduce 框架的局限性。若计算任务易于拆分成一组针对（Key，Value）的操作，那 MapReduce 将能充分发挥作用，否则还是选取其余并行计算框架进行处理为宜。

3.2.3.2 Hadoop MapReduce

Apache Hadoop 项目中包含了一份 MapReduce 开源实现。在实际 MapReduce 程序开发过程中，开发人员除了 Map、Reduce 函数实现还需要提供一些额外信息。具体如图 3.4 所示。

```
Job Configuration // 设置MapReduce任务各项参数

InputFormat     // 提供输入数据划分策略
InputSplit      // 表示每一个数据划分块
RecordReader    // 从InputSplit中读取记录

Input Location  // 指定输入数据所在路径

OutputFormat    // 提供数据输出策略

Output Location // 指定MapReduce输出路径

Mapper          // 提供Map函数实现

Reducer         // 提供Reduce函数实现
```

图 3.4 MapReduce 任务配置

图 3.4 中标记的是常见的 MapReduce 任务需要的配置，某些具体场景下并不需要提供上述所有的配置，如 MapReduce 任务可以在 Reducer 阶段选择直接输出

数据而非通过 OutputFormat 来完成输出。此外 MapReduce 框架也允许开发人员进行更为详细的配置信息进行性能调优。MapReduce 任务的执行步骤如图 3.5。

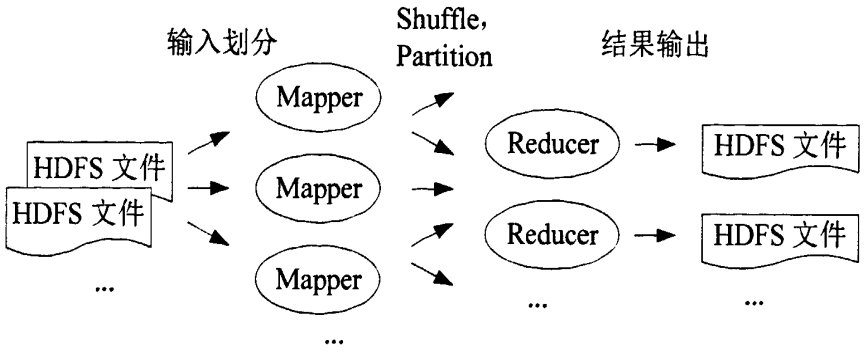


图 3.5 MapReduce 执行过程

3.3 系统设计

在了解了系统选取的各项关键技术后，我们可以根据原型系统的设计目标细化整体设计方案。

3.3.1 整体架构

原型系统主要包含两个部分：客户端模块以及集群端模块。原型系统的整体架构如图 3.6 所示。

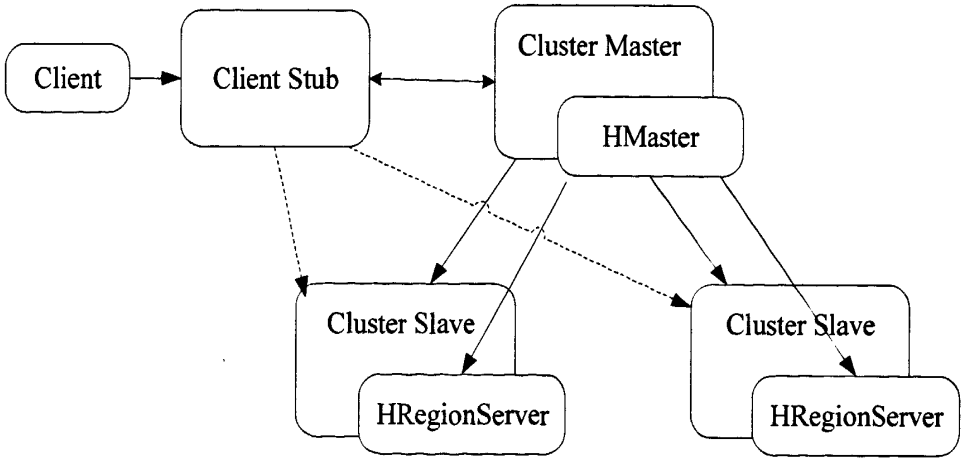


图 3.6 原型系统架构

原型系统的集群端模块也实现了类似 Master-Slave 的管理模型。系统的

Master 节点与 Slave 节点在部署上同 HBase 集群的 HMaster、HRegionServer 一一对应。在原型系统的设计中考虑了 HBase 集群的对于底层硬件可靠性的需求。HBase 集群的 Master-Slave 模型中，HMaster 节点负责管理维护大量元数据信息，若 HMaster 节点发生异常则会严重影响整个集群功能甚至导致集群无法恢复工作。所以 HMaster 所在的机器对硬件性能以及可靠性的要求都更高。因此在选择 Master 节点所在时也需考虑底层硬件在这方面的需求。

原型系统的客户端模块当前主要负责同集群节点的交互。客户机器存储数据均通过客户端模块进行，客户端模块对外隐藏了后端集群通信的具体细节。

3.3.2 功能模块

当前原型系统主要提供的功能包括：RDF 数据存储、SPARQL 查询处理。RDF 数据文件从客户端机器传输到集群节点并加载到 HBase 当中，在 RDF 数据加载完成之后，集群对外提供 SPARQL 查询支持。原型系统对客户端请求的应答步骤如图 3.7 所示。

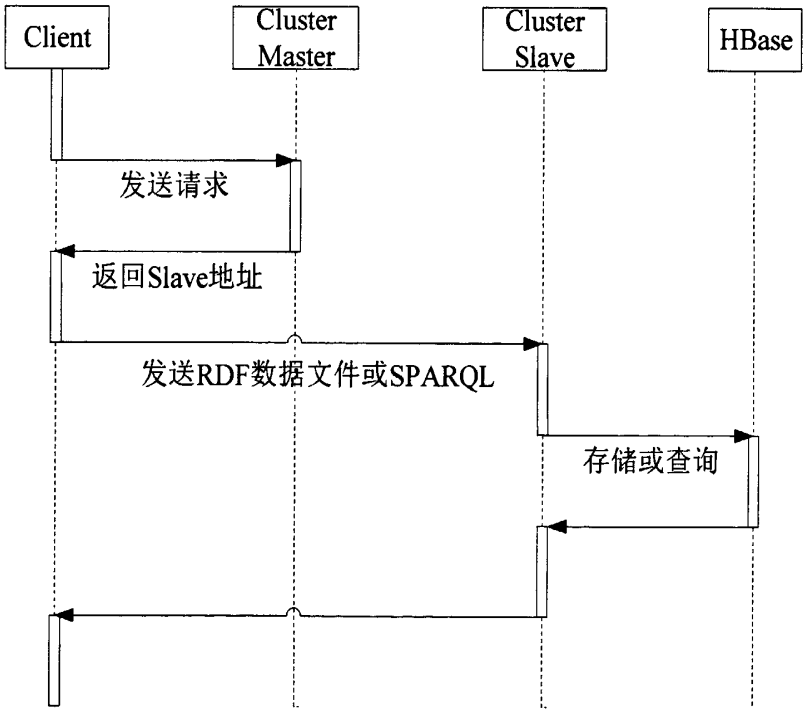


图 3.7 原型系统请求响应序列图

原型系统提供的RDF存储与查询功能均以Slave节点直接负责响应客户端请求。客户端在发起操作时首先通知 Master 节点，随后 Master 节点返回客户端告知其具体应当连接哪个 Slave 节点。随后客户端将直接同 Slave 节点进行通信与数据传输。

在 RDF 数据存储方面，客户端将发送一系列 RDF 数据文件到 Slave 节点，Slave 节点随后进行一系列处理并将其加载到 HBase 当中。RDF 数据在 HBase 中的存储结构以及 HBase 数据导入过程的详细步骤将会在第四章中进行具体介绍。

在 SPARQL 查询响应方面，客户端将发送 SPARQL 查询语句到 Slave 节点，Slave 节点随后通过 ARQ 解析查询语句并生成具体查询计划用以响应请求。查询计划生成以及查询处理的策略将在第五章中进行具体介绍。

3.3.3 通信模型

图 3.7 给出了原型系统同客户端间的通信步骤，系统 Master 节点负责管理分配 Slave 节点进行任务响应。因此 Master 节点需维护各 Slave 节点的运行状况。考虑到洗头构建在 HBase 集群之上，因而 Master 节点与 Slave 节点的状态信息可以存储在 HBase 数据表当中。Master 节点通过访问 HBase 数据表中记录的节点状态信息可以掌握当前系统的运行状态。在 HBase 中的节点状态信息表定义如图 3.8 所示。

```
Row-Key: IP-PC-NAME
  Column Family: Meta
    Column: Role       // 标记该节点的角色
      T1 Master
    Column: Heartbeat // 定期更新标记存活状态
      T2 Value1
    Column: IsBusy     // 记录节点是否在响应任务
      T3 True
    Column: IsDown     // 标记节点是否不能工作
      T4 False
```

图 3.8 集群节点信息表

Master 节点以及各 Slave 节点均需在一定时间间隔内去上述状态信息表中更

新其对应的行记录。HBase可以记录不同时间戳下的多个数据版本，故而Master节点可以通过读取Heartbeat字段更新的时间来判断节点的存活状态。

3.4 本章小结

本章首先描述了基于HBase的RDF存储系统原型的设计目标。随后本章对构建系统所需的几项关键技术HBase、ARQ、MapReduce进行了详细的介绍。最后本章阐述了原型系统的设计思路与架构，介绍了当前原型系统提供的RDF数据存储与SPARQL查询处理的响应流程。本章在整体层面详细介绍分析了原型系统的设计方案后勾勒出了一个基于HBase的RDF存储系统的雏形。

第4章 RDF存储模型定义

在上一章对原型系统整体架构进行介绍后，本章将具体介绍 RDF 数据具体如何在 HBase 上进行存储。本章首先分析了前文描述的几种经典 RDF 存储模型应用在HBase上的优缺点，经过对不同存储模型的分析我们可以发现RDF数据在HBase 上存储的关键问题在于如何对稀疏的 RDF 三元组构建有效的索引结构。在明确了索引构建这一问题后，本章提出了利用多个 HBase 数据表进行 RDF 数据存储索引的方案。RDF 三元组将被存放在 SPO、POS、OSP 三张数据表当中。三张表运用了 HBase 提供的针对 Row-Key 的默认索引结构来存储索引 RDF 数据。RDF 数据存储于 HBase 中的另一问题是 RDF 数据初始导入的效率问题，本章结合 HBase 当前包含的 Bulk Load 功能描述了 RDF 数据导入的具体流程。

4.1 HBase 上 RDF 存储模型分析

本文第二章介绍了几种经典的 RDF 存储模型：Triple Store、Property Table、Vertical Partitioning、Hexastore。本小节将分别评估上述模型应用于 HBase 之上的具体方案及优缺点。

4.1.1 应用 Triple Store

Triple Pattern 在关系数据库中只需要一张数据表，不同三元组分别插入不同数据行当中。但在 HBase 中难以采用直接采用 Triple Pattern 来存储 RDF 三元组。

在 HBase 中应用 Triple Pattern 的主要问题在于难以选定 Row-Key。RDF 数据中存在大量 Subject 相同的三元组，但在 HBase 中每一行的 Row-Key 需要是唯一的，因此难以做到完全符合 Triple Pattern 描述的表结构定义。

4.1.2 应用 Property Table

Property Table 在关系数据中应用时受限于数据表的列数限制，因此需要引入 Cluster 算法来决定如何将不同属性分别聚集。但 HBase 中并不存在列数限制，

HBase中一行数据可以有任意多列，数据表的宽度不再成为问题。

在HBase上应用Property Table的方案大致如下。HBase内需创建一张数据表，表的Row-Key选择RDF三元组的Subject，表中包含一个Column Family：Predicates。在这唯一的Column Family下将存储该Subject包含的所有属性名称以及属性值。

此种方式的优点在于：

- 表结构简单。HBase内用于存储RDF数据的只有一张表，且该表中只有一个Column Family。
- 避免空值。HBase是基于Column Family进行存储的，不同行中的Column Family可以包含完全不同的Column，这种存储形式避免了在基于行的关系数据库中的空值问题。
- 支持Multi-Valued属性。同一个Subject的所有属性均存储在同一个Column Family下的不同行中。Multi-Valued属性也可以不作区分进行存储。如[25]中所述，Multi-Valued属性可以以多种形式存储，譬如数组形式。
- 避免自连接操作。同一个Subject对应的属性都存储在同一行当中，因此在读取数据行时可以直接采用HBase提供的Filter功能从而避免进行自连接操作。

此种方式的缺点在于：

- 索引结构难以创建。HBase默认只对Row-Key进行索引，也就是说在上述表设计中，HBase只能够快速响应Subject确定的Triple Pattern，如（S, ?P, ?O）、(S, ?P, O)、(S, P, ?O)。对于Subject未知的查询来说默认只能通过遍历整张数据表来进行查询，在海量数据下这是难以接受的。HBase提供了支持额外索引的IndexedTable实现，但创建额外索引需指定针对哪些Column创建。在Property Table模型中，每一个不同属性都需要对应的索引，这样才能高效的响应所有可能的Triple Pattern。但这会引入过多的索引，索引创建以及索引维护上的开销都难以接受。

4.1.3 应用 Vertical Partitioning

在 HBase 中应用 Vertical Partitioning 可以像在关系数据库中那样针对不同属性创建不同的表，每张表中用 Subject 作为 Row-Key，Object 存放在唯一的 Column Family 当中。此种方式的优缺点同其应用在关系数据中类似。

另外一种在 HBase 上应用 Vertical Partitioning 的方法是创建一张数据表。表的 Row-Key 选用 Subject，且表中包含大量不同的 Column Family，每一个 Column Family 对应一个属性。

此种方式的优点在于：

- 避免创建大量数据表。HBase 当前对表的管理方式十分简单，数据表不像关系数据库中那样可以通过 Schema 等机制进行管理。因此过多的数据表会造成维护上的困难。
- 易于创建额外索引表。在上述结构设计下，系统可以再创建另外一张表来冗余存储数据。表的 Row-Key 选用 Object，表中为每一个属性创建一个 Column Family，Column Family 下存储 Subject。在响应 Triple Pattern 时，可以根据 Subject、Object 是否绑定来选择用哪张表进行响应。

此种方式的缺点在于：

- 难以快速响应（?S, P, ?O）查询。如若 Subject 与 Object 均未指定，那么在上述设计中无法快速找到对应的数据行。为了响应该查询，系统需要对数据表进行遍历，这在性能上也是难以接受的。

4.1.4 应用 Hexastore

Hexastore 不同于上述三种模型之处在于其并非直接构建在关系数据库表结构之上，所以应用到 HBase 之上需要进行一定转化。针对 Hexastore 提出的六个索引，在 HBase 中可以创建六张表来对应：S_PO、P_SO、O_SP、SP_O、SO_P 与 PO_S。

具体表结构定义如下：在 S_PO 表中 Subject 作为 Row-Key，包含一个 Column Family，其下每一个 Column 分别存储不同的(Predicate, Object)，值存放在 Column

中。在 SP_O 表中（Subject, Predicate）作为 Row-Key，包含一个 Column Family，其下每一个 Column 分别存储不同的 Object。

此种方式优点在于：

- 无需额外索引结构。上述六张表就已经相当于提供了不同的索引，因此无需再创建其它索引结构。这里充分利用了 HBase 针对 Row-Key 提供的索引能力。
- 快速响应 Triple Pattern。每一个 Triple Pattern 对应的结果均存储在不同数据表的一个数据行之上，因此 Triple Pattern 的查询效率能够能到保证。

此种方式缺点在于：

- 过多的额外存储空间。同 Hexastore 索引类似，这种方案需要大量额外的存储空间。不同于 Hexastore 的优化策略，这里不同表中的数据无法被重用。因此额外的空间开销不可避免，大体上来说最终需存储的数据约是原始数据的六倍。
- 数据表过宽。上述数据表面临数据过宽的问题。特别是 P_SO 表中，一行中可能会存在数百万列，譬如针对 rdf:type 属性。HBase 虽然支持任意数目的列数，但当前它实现的 API 接口将会返回一行中的所有数据。因此如若数据行变得很大，那么返回的数据大小可能会超出当前的内存设定。因此存在潜在的问题。

4.2 HBase 上 RDF 存储模型定义

在进行了上述分析之后，我们可以看出在 HBase 之上 Triple Store 并不适应；Property Table 则受困于额外索引结构的创建问题；Vertical Partitioning 不能对所有的 Triple Pattern 进行很好的响应；Hexastore 能够快速响应 Triple Pattern，但也存在一些问题。

HBase 中 Row-Key 的查询效率是通过其默认索引结构保证的，额外索引（Secondary-Index）的创建是一个巨大的开销。考虑到索引创建这一问题，利用 HBase 默认提供的针对 Row-Key 的索引是保证数据查询效率的关键。本文最终选

取的RDF存储模型构建在前文描述的基于Hexastore方案之上，但我们对上述方法进行了进一步优化，减少了所需的额外存储空间并规避了数据表过宽的问题。

4.2.1 RDF数据表定义

在经过详细的分析后，我们发现采用六张HBase数据表对应Hexastore的六个索引结构有些冗余。在本文选定的RDF存储方案中，RDF三元组将存储于三张数据表：SPO、POS与OSP。三张数据表在存储RDF数据时也发挥了索引结构的作用。

SPO、POS与OSP三张表的表定义相同，每张表只包含一个Column Family，每一行的数据均存储在着一个Column Family中。三张表的不同之处在于存放在其中数据不同。SPO表的Row-Key是（Subject，Predicate），Column Family中存放Object值；POS表的Row-Key是（Predicate，Subject），Column Family中存放Subject值；OSP表的Row-Key是（Object，Subject），Column Family中存放Predicate值。具体如图4.1所示。

```
Row-Key: (Subject, Predicate) {
  Column Family: Objects {
    Column Object1
    Column Object2
  }
}
```

(a) SPO表

```
Row-Key: (Predicate, Object) {
  Column Family: Subjects {
    Column Subject1
    Column Subject2
  }
}
```

(b) POS表

```
Row-Key: (Object, Subject) {
  Column Family: Predicates {
    Column Predicate1
    Column Predicate2
  }
}
```

(c) OSP表

图 4.1 SPO、POS、OSP表结构定义

4.2.2 Triple Pattern查询响应

在响应Triple Pattern查询时，若Triple Pattern中Subject、Predicate、Object

中任意两个值绑定，则该 Triple Pattern 可以直接提取绑定的值来生成对应 SPO、POS、OSP 三张表的 Row-Key，再从对应的数据表中读取记录。若 Triple Pattern 中 Subject、Predicate、Object 中只有一个值绑定，则需要进一步分析此时 HBase 中数据的存储情况来决定如何进行查询响应。

图 4.2 展示了 SPO 表中存储的数据，从中可以看出 HBase 数据表中 Row-Key 的存储是有序的。假设 Triple Pattern 中只有 Subject 绑定，那么在进行查询时可以通过 HBase 提供的 Scan 进行区域查询。Scan 区域查询需要提供起始与终止 Row-Key，这里只需要提供 Subject 值加上分隔符即可。由于 Row-Key 的有序性，Scan 的查询效率可以得到保证。

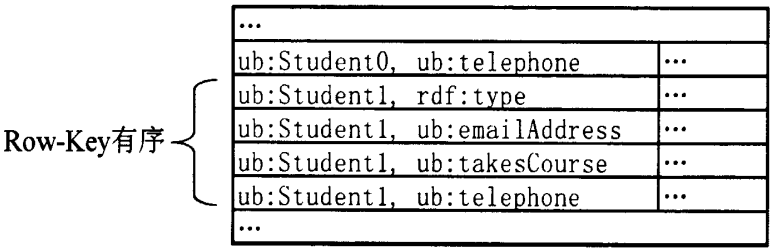


图 4.2 SPO 表

根据上面的描述，Triple Pattern 与 SPO、POS、OSP 三张表间的对应关系如表 4.1 所示：

表 4.1 Triple Pattern 映射关系

Triple Pattern	Table
(S P O)	任意
(S P ?O)	SPO 表
(S ?P O)	OSP 表
(S ?P ?O)	SPO 表
(?S P O)	POS 表
(?S P ?O)	POS 表
(?S ?P O)	OSP 表

(?S ?P ?O)	任意

在上述方案中，我们可以只通过三张数据表实现 Hexastore 描述的六个索引的功能，同最初的基于 Hexastore 方案相比，该方案节省了近一倍的存储空间。

4.2.3 数据多行切分

同样根据 HBase 对 Row-Key 有序存储这一特性，当数据表列过多时，可以将一行的数据拆分到多个数据行当中，进行 Fat Table 到 Tall Table 的转换。HBase 虽然支持任意数目的列，但是列数过多并不利于数据的划分。此外当前 HBase 版本在进行数据读取时会一次性返回整行的结果。若该行的数据过大，则可能会因为内存不足而导致程序在进行数据读取时遭遇内存不足的异常。图 4.3 展示了用多个数据行存储的示例。同一个类型的 RDF 三元组可能存在数百万条记录，在这种情况下就可以对其进行拆分。在进行查询时依然可以通过 Scan 获取到临近的区域。

…	
rdf:type, ub:Student, split1	…
rdf:type, ub:Student, split2	…
rdf:type, ub:Student, split3	…
rdf:type, ub:Student, split4	…
rdf:type, ub:Student, split5	…
…	

图 4.3 多行拆分示例

4.3 RDF 数据导入

在确定了 HBase 上的 RDF 存储模型之后，这一节将介绍如何将 RDF 数据导入到 HBase 数据表当中。RDF 数据的导入过程分为如下两步：将 RDF 数据文件导入 HDFS 分布式文件系统，将 HDFS 上的 RDF 数据文件导入 HBase 之中。

4.3.1 RDF 数据导入 HDFS

海量 RDF 数据可能包含上百万上千万 RDF 三元组，为了将它们高效导入 HBase 之中，在当前集群环境下理应采用 MapReduce 并行处理。MapReduce 能够操作的数据必须存放在 HDFS 分布式文件系统当中，因此首先需要将 RDF 数据

文件存放到HDFS中。

HDFS不适合存放大量小文件，过多的小文件会产生大量元数据性喜从而给HDFS主节点带来过重的负担。而RDF数据往往以大量小文件形式存在，因此在进行导入时需要将多个文件进行合并。同时为了考虑后续操作的便利性，RDF数据文件合并的结果将同上文定义的SPO、POS、OSP表相对应。

4.3.2 RDF数据导入HBase

RDF数据在存储到HDFS上之后任需要进一步存储到HBase当中。这一步唯一需要考虑的就是数据导入的性能问题。

HBase的org.apache.hadoop.hbase.client包下提供了HTable类用于存取HBase中的数据。常见的数据导入方式就是通过调用HTable的API进行处理。但此种方式的性能往往很难保证。此外HBase在对数据行进行写操作时会对整个行进行加锁，在导入大量RDF数据时可能存在大量的写冲突，需要很多额外的等待锁释放锁操作。虽然可以通过一定预处理操作以及调整HBase参数配置来获得性能提升，但这种方法不具备良好的通用性。

HBase底层数据存放在HDFS之上，因此如若可以绕开HTable接口直接创建HBase数据文件，那么将会大幅度提升RDF数据导入性能。在查阅了HBase相关资料后，我们发现当前HBase提供了Bulk Load工具进行批量数据加载。因此本文最终选用了HBase提供的Bulk Load进行RDF数据加载。Bulk Load主要包含两个步骤：HBase数据文件生成、HBase数据文件加载。

HBase数据文件生成需要通过一个MapReduce任务来实现。该MapReduce任务的输入是存放在HDFS之上的RDF数据文件，输出是HBase数据文件。在Map阶段，Mapper将输入的文本数据组装成HBase的Result对象。在Reduce阶段，Reducer将对同一个Row-Key下的多个Result对象进行排序。Reducer采用的是HBase提供的KeyValueSortReducer，HBase数据文件生成则交由HBase提供的HFileOutputFormat。在应用HFileOutputFormat时需要保证各个Reducer接收到的数据在全局是有序的，因此在Reducer数目大于1时，该任务就不能够采用

MapReduce 框架默认的 HashPartitioner 进行 Mapper 的输出数据的划分。Hadoop 提供了 TotalOrderPartitioner 实现来确保数据划分的有序性，TotalOrderPartitioner 需结合 SequenceFile 来确定每个 Reducer 接收的数据范围。在应用 TotalOrderPartitioner 时要提前通过 InputSampler 生成 SequenceFile。此外还需将 MapReduce 的 mapred.reduce.tasks.speculative.execution 参数设置为 false。该任务的关键配置如图 4.4 所示。

```
MapReduce Configuration:
 Partitioner:  TotalOrderPartitioner （配合InputSampler使用）
 Reducer:      KeyValueSortReducer
 OutputFormat: HFileOutputFormat

MapReduce参数设置:
 // 一个Reduce Task只交由一个Reducer执行
 mapred.reduce.tasks.speculative.execution : false
```

图 4.4 RDF 数据导入任务参数配置

HBase 数据文件的加载需要通过 loadtable 工具。Loadtable 工具可以直接将上述 MapReduce 的输出加载到 RegionServer 中，从而实现数据的批量加载。

4.4 本章小结

本章首先分析了在 HBase 之上应用各种 RDF 数据模型的方案以及优缺点，在评估了各种方案的优劣后提出了本文采用的 RDF 存储模型，即通过 SPO、POS、OSP 三张数据表来存储索引 RDF 三元组。SPO、POS、OSP 三张数据表可以满足所有不同的 Triple Pattern 查询，在保证 Triple Pattern 查询性能的同时减少了 RDF 数据的数据存储空间。在定义了 RDF 数据模型之后，本章介绍分析了 RDF 数据加载到 HBase 所需考虑的问题并提出了本文选用的具体的 RDF 数据导入方案。针对本文选用的 HBase Bulk Load 方案，本章详细介绍了数据导入的各个步骤，分析了每个步骤需考虑的问题并给出了所需的关键配置。

第5章 RDF 查询处理

本章主要关注的是 RDF 标准查询语言 SPARQL 中 BGP 子句的连接处理。在海量 RDF 数据背景下，BGP 中的连接操作涉及了海量数据集合之间的连接处理。在这种情况下，本文选用了通用并行计算框架 MapReduce 作为查询响应工具。

MapReduce 并行框架在同一个时间只能处理一个任务，且集群对于每个任务都需要一定时间进行初始化操作。因此在采用 MapReduce 处理 BGP 连接时需要控制任务数目以提高集群资源利用率。据此本章提出了在考虑降低 I/O 开销的基础之上进行 MapReduce 多路连接选择的决策方法。此外本章也描述了针对单一 MapReduce 连接处理任务的优化方法，提出了在 Mapper 阶段进行数据过滤以降低无用数据从 Mapper 到 Reducer 之间网络传输开销的处理策略。

5.1 SPARQL 预处理

本文第二章中已经对 SPARQL 进行过介绍，SPARQL 语句参见图 2.1。SPARQL 查询预处理包含两个步骤：SPARQL 查询解析、查询计划生成。

5.1.1 SPARQL 查询解析

本文选用了开源工具 ARQ 来进行 SPARQL 查询解析。从图 5.1 可知，程序可以从 SPARQL 解析结果直接获取到原查询中的 BGP 块以及 BGP 中的各 Triple Pattern 子句。后续查询计划生成以及各项操作都基于 ARQ 的解析结果。

```
SELECT ?x WHERE {
  ?x rdf:type ub:GraduateStudent .
  ?x ub:takesCourse GraduateCourse0
}
```

(a) LUBM Q1

```
(project (?x)
  (bgp
    (triple ?x rdf:type ub:GraduateStudent)
    (triple ?x ub:takesCourse GraduateCourse0)
  )
)
```

(b) ARQ解析结果

图 5.1 SPARQL 查询解

5.1.2 查询计划生成

ARQ 的解析结果展示了 Triple Pattern 子句的详细信息。Triple Pattern 间的连接关系可以通过分析各 Triple Pattern 中未绑定变量的名称获得。图 5.2 展示了常见的 Triple Pattern 连接方式。

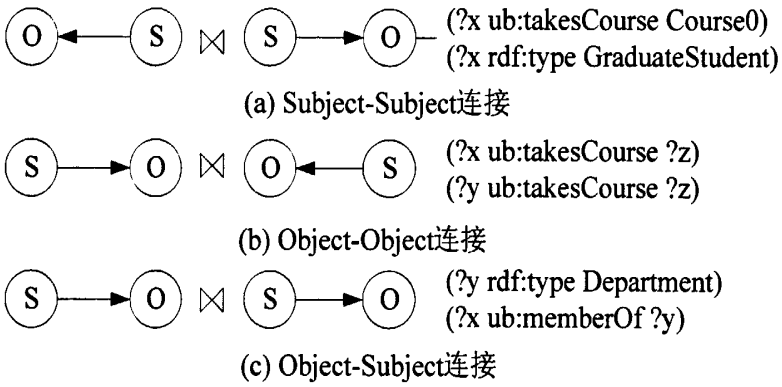


图 5.2 Triple Pattern 连接方式

Triple Pattern 间的关联性是通过它们之间是否存在同名的未绑定变量判定的。根据同名变量出现的位置可以将它们间的连接划分为：Subject-Subject 连接、Object-Object 连接、Object-Subject 连接三种。真实 BGP 中的连接大都由上述三种连接方式混合而成，所以在使用 MapReduce 处理连接时需多个任务迭代完成。

为了有效的控制 MapReduce 的任务数目以及每个迭代过程中任务的复杂性，本文提出了基于[21]中描述的多路连接选择策略。[21]介绍了在一个 MapReduce 任务中进行多路连接的思路，但未说明如何确定多路连接查询计划。本文在考虑各 Triple Pattern 间的关联关系以及参与连接的各数据集的大小之上提出了基于贪心的查询计划生成策略。这里以图 5.3 中的 LUBM Q8 为例具体介绍具体步骤：

1) 解析 BGP 获取所有查询变量，并统计各查询变量出现的次数。
2) 贪心选取查询变量用于选择进行连接的数据集。首先选取出现次数最多的查询变量。在出现次数相同情况下优先选择出现在高选择性子句中的变量，这里的选择性排序如下：如果参与连接的数据集是 Triple Pattern，则 Subject 绑定的子句选择性最高，Predicate、Object 绑定的子句选择性次之（Predicate 非 rdf:type）；如果参与连接的是 HDFS 上的中间数据，

则选择性根据其数据大小从小到大排序；Triple Pattern 与中间文件之间则认定 Triple Pattern 选择性更高。

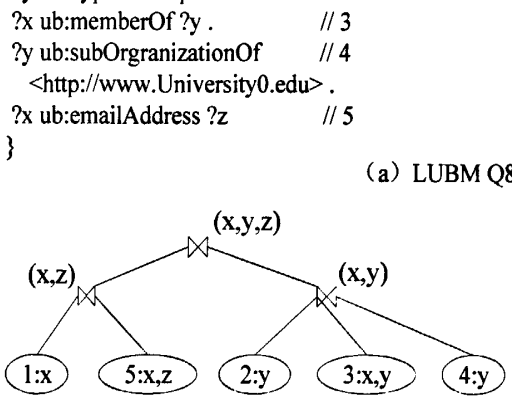
3) 根据上一步选定的顺序，依次连接拥有相同变量的数据集。如图 5.3（b）所示变量 y 在出现次数与变量 x 相同情况下由于其出现在高选择性子句 Triple Pattern 4 中，因此优先选取变量 y 进行连接划分。

4) 查看最终生成的查询计划中是否存在多路连接，在保证最终查询树深度不变情况下按顺序尝试是否能消除多路连接。图 5.3（b）中所示结果包含了 Triple Pattern 子句 1 与 5 之间的连接，但这一连接不是必须的，移除这一连接之后查询树高度不会改变。图 5.3（c）展示了最终的查询。与图 5.3（b）相比图 5.3（c）中的查询计划在 MapReduce 任务数目一致的情况下避免了 Triple Patter 子句 1 与 5 的数据读取与中间结果生成。

```
SELECT ?x ?y ?z WHERE {
 ?x rdf:type ub:Student .          // 1
 ?y rdf:type ub:Department .       // 2
 ?x ub:memberOf ?y .               // 3
 ?y ub:subOrgranizationOf          // 4
   <http://www.University0.edu> .
 ?x ub:emailAddress ?z             // 5
}
```

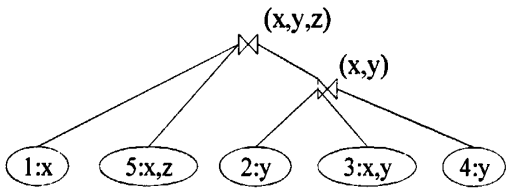
分析：
查询变量统计：(x,3),(y,3),(z,1)
最高选择性子句：Triple Pattern 4

（a）LUBM Q8



分析：
双层查询树
第一个MapReduce任务分别根据x，y划分

（b）优先选取y的查询树



分析：
双层查询树
第一个MapReduce任务只根据y划分

（c）去除冗余多路连接后的查询树

图 5.3 查询计划生成

5.2 BGP 连接处理策略

在明确查询计划之后，本小节将介绍在每一个 MapReduce 任务中如何具体进行连接操作。连接操作中的主要问题有：如何协同读取划分 HBase 中数据与 HDFS 中中间结果；如何定义 Mapper、Reducer 输入输出进行多路连接；如何在 Map 阶段优先过滤数据；如何在 Reduce 阶段进行最终连接。本小节将依次对上述问题进行说明并给出解决方案。

5.2.1 HBase 数据读取

当 Triple Pattern 第一次参与连接操作时需要首先将数据从 HBase 数据表中读取出。第四章中介绍了 RDF 数据是如何在 HBase 上进行存放，对于每一个 Triple Pattern 子句来说，它对应的结果存储在 SPO、POS 或是 OPS 中的某一张表的连续数据区域当中。Triple Pattern 与这三张表的对应关系已经在表 4.1 中进行过描述。在选定数据表后，HBase 提供了 Scan 用于数据访问。Scan 类需要的数据起始与终止区域的 Row-Key 可以通过分析 Triple Pattern 获取到。

根据第三章对 HBase 的介绍，HBase 表中的数据可能根据大小分裂成多个 Region，不同的 Region 分属不同的 RegionServer 管理。在进行 MapReduce 连接操作时划分 Triple Pattern 结果相当于划分 HBase 数据表中的连续区域。这里的划分策略如图 5.4，每个 Triple Pattern 结果划分的份数等于其对应数据区域跨越的 Region 数目。当连接操作包含多个 Triple Pattern 时，总的划分数目等于每个 Triple Pattern 结果划分数目总和。

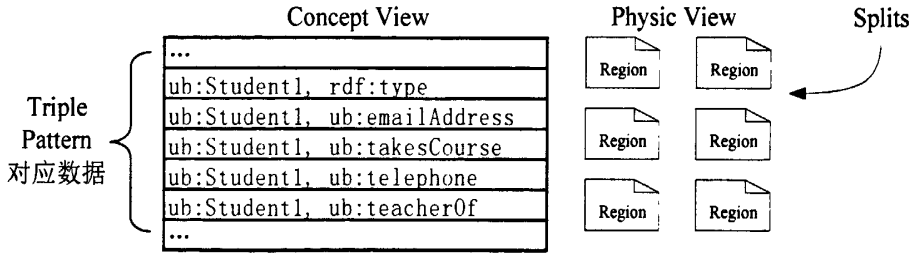


图 5.4 HBase 数据划分

HBase 中每个 Region 的在超过了预设的大小之后就会进行分裂，Region 的

分裂操作由HBaseMaster进行管理。由于无法预先得知Triple Pattern结果区域中间可能出现的哦Row-Key值，Triple Pattern结果的划分数目只能同其所跨越Region数目相同。当Triple Pattern对应结果越多时，划分的数目也越多。当结果数据只有少数条目时，其往往只存在于单一的Region当中，因而只有在海量数据下这样的划分策略才能最好的发挥作用。

5.2.2 HDFS数据读取

在采用多个MapReduce任务进行BGP连接操作时，前一个MapReduce任务的输出作为后一个任务的部分输入，两者之间的数据交互只能完全通过HDFS分布式文件系统。因此在进行连接响应时不得不同HDFS进行交互。

在中间结果的存储上，本文选择直接将其存储在HDFS之上而非HBase之中。HBase之中的数据存放是有序的，但对中间结果来说有序性并不重要。将它们直接存放在HDFS上更加简单高效。中间结果将以文本文件格式写入HDFS上的制定文件夹当中。值得注意的一点是在进行多路连接时，MapReduce任务可能需同时对多个不同数据集合同时进行连接。因此为了保证后续操作的有效性，每个连接的结果必须输出到不同目录当中。

在中间结果读取方面，MapReduce任务将会从指定的路径进行读取。针对输入的划分策略将采用Hadoop默认提供的FileInputFormat中的策略。

5.2.3 Mapper输入输出格式

为了完成设定的MapReduce任务，我们需实现自定义的InputSplit、RecordReader。在具体进行相应时，每个Triple Pattern以及中间结果都会被分配一个唯一的ID标识，InputSplit中要保存这个序列号以获知数据来源。RecordReader可以访问InputSplit中的数据来源ID，针对不同的来源（HBase数据表、HDFS文件）进行对应的格式转化。每个数据来源对应的变量信息存放在全局可见的Job Configuration当中。最终Mapper接收的（Key，Value）值中的Key既是数据来源ID，Value则是每一条记录的值。Mapper的输出需要保证按照选定的变量进行划分，因此Mapper输出的Key中要包含对应的变量名，Value中

要包含数据来源ID。输入输出格式如图5.6所示。

Mapper Input:
(2, GraduateCourse1)
(2, GraduateCourse16)
(3, UnderGraduateStudent525, GraduateCourse16)
...

Mapper Output:
((x,GraduateCourse1), (2, GraduateCourse1))
((x, GraduateCourse16), (2, GraduateCourse16))
((x, GraduateCourse16), (3, UnderGraduateStudent525, GraduateCourse16))
...

图 5.5 Mapper输入输出格式

5.2.4 Map阶段过滤操作

Map阶段的主要任务是进行输入数据的重新组装，连接操作的实现在Reduce阶段。在这种连接方式处理下，所有的数据都将会被 Mapper 读入再被发送给Reducer。即便大部分数据不会出现在最终连接结果当中，它们也依然要经历上述转发步骤。大量数据在MapReduce框架的传递是需要消耗一定资源，因此出于性能考虑应当尽可能减少Mapper的输出。

如果Mapper可以完全将部分数据集合加载到内存当中，那么在Mapper阶段提前对数据进行过滤就成为可能[26]。在MapReduce任务提交之前，系统可以根据每个Triple Pattern子句或是中间结果对应的数据大小判定其是否能够加载到内存当中。而后在Map阶段将据此进行判定过滤。过滤判定可以采用Hadoop提供的BloomFilter实现。在参与连接的多个数据集之中程序将选取一个或多个进行创建BloomFilter。对于HBase中的数据，我们根据Triple Pattern的类型大致估计最终结果数目。程序将选取Subject绑定或Predicate与Object同时绑定（Predicate非rdf:type）的 Triple Pattern 子句。对于 HDFS 中的数据，程序将读取前一次MapReduce任务的Counter信息以获得输出大小。生成的BloomFilter对象将会被存放在Hadoop提供的DistributedCache中用于被MapReduce框架在运行期访问。

5.2.5 Reduce 阶段连接操作

根据前文描述的 Mapper 输出格式定义，Reducer 在收到多个 Mapper 的输出后依然可以判定输入的初始来源是哪一数据集，当前数据是依照哪一查询变量进行划分。Reducer 的输入是（Key, List<Value>），首先 Reducer 将读取链表中的所有 Value 并解析，将不同来源的数据存放在一起；而后 Reducer 将对不同来源数据进行排序；最终将对多个来源的数据进行连接。

5.2.6 连接处理完整流程

在上文对 MapReduce 连接处理进行详细描述后，其完整处理流程如图 5.6 所示。在提交 MapReduce 任务之前，程序将判定哪些数据集可以用于创建 Filter。而后将按照标准的 MapReduce 任务进行数据划分与 Map、Reduce 两阶段处理。

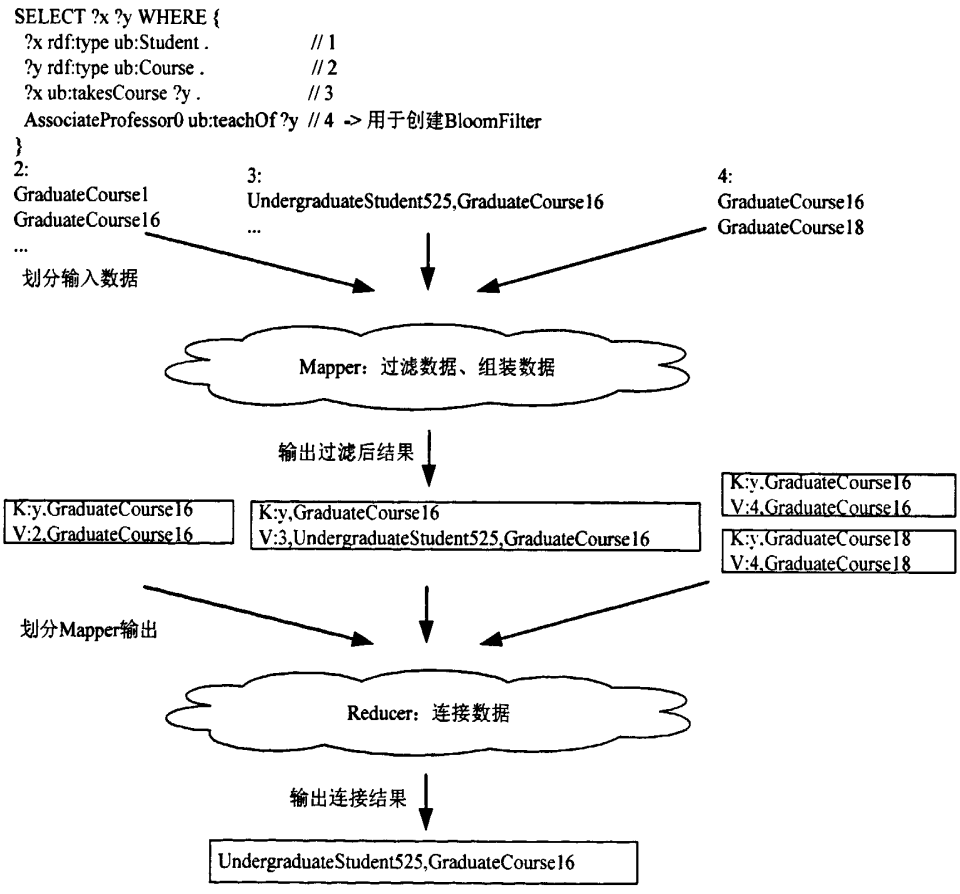


图 5.6 MapReduce 连接处理流程

5.3 本章小结

本章首先叙述了对SPARQL的解析处理，而后描述了在ARQ解析结果上的查询计划生成算法。本章在[21]基础上提出了多路连接选择判定方法，在进行多路连接判定时优先考虑各连接集的大小，小数据集进行连接时可以进行Map阶段过滤等优化操作以避免MapReduce任务中过多的I/O开销。随后本章详细描述了单个MapReduce连接任务涉及的各个步骤，分析了从HBase、HDFS上读取数据的方式以及数据划分策略，定义了Mapper的输入输出格式以及在Map阶段进行BloomFilter过滤的策略。最后本章介绍了Reducer内进行连接的方法并给出了完整的MapReduce连接处理流程。

第6章 测试与实验

本章对前文提出的 RDF 查询处理方法采用 Lehigh University Benchmark（LUBM）标准测试数据集进行了实验验证。本章测试了五组不同 LUBM 数据集下的 RDF 查询响应情况。

6.1 实验环境

实验环境如表 6.1 所示，其中 HBase 集群中 HRegionServer 所在机器的物理内存均为 4GB，分配给 HBase 的内存为 1GB。

表 6.1 实验环境

HBase 集群数目	8
节点处理器	Intel E8400 3.00GHz
节点内存	4 GB

实验数据采用了 LUBM 测试数据集，LUBM 测试查询中的部分查询语句需要查询处理引擎具备在 subPropertyOf、subClassOf 关系上的推理能力，然而当前原型系统并不支持对 RDF 数据进行推理。为了让这些查询语句能够有效工作，我们需要获取到 LUBM 数据上针对 subPropertyOf、subClassOf 关系的推理结果。在进行测试时，我们通过 Jena RDFSRuleReasoner 处理 LUBM 数据集以获得最终测试数据。

6.2 RDF 查询测试

我们针对如下几组不同大小的 LUBM 数据集进行测试，数据大小如表 6.2 所示。表 6.2 中的 RDF 三元组是经过 Jena RDFSRuleReasoner 处理后的数目。

表 6.2 测试集大小

Universities	1	50	100	300	500
RDF Triples	151,405	10,108,912	20,347,317	60,550,872	101,314,243

我们在不同数据集下分别测试了八条 LUBM 查询语句，各查询的响应时间如表 6.3 所示。

表 6.3 LUBM 查询响应时间

Univ.	Q1(sec)	Q2(sec)	Q3(sec)	Q4(sec)	Q5(sec)	Q6(sec)	Q7(sec)	Q8(sec)
1	20	46	23	24	21	27	47	49
50	27	80	27	61	27	47	81	113
100	27	118	35	100	31	59	99	157
300	35	215	47	120	39	79	185	243
500	45	327	58	136	42	106	210	249

6.2.1 LUBM Q1、Q3、Q5 查询结果分析

LUBM Q1、Q3、Q5 三条查询语句类似，每个查询均只包含两个 Triple Pattern 子句，其中一个 Triple Pattern 子句具备高选择性。三条查询语句如图 6.1 所示。

```
SELECT ?x  WHERE {
 ?x rdf:type ub:GraduateStudent .
 ?x ub:takesCourse
   http://www.Department0.University0.edu/GraduateCourse0
}
```

(a) LUBM Q1

```
SELECT ?x  WHERE {
 ?x rdf:type ub:Publication .
 ?x ub:publicationAuthor
   http://www.Department0.University0.edu/AssistantProfessor0
}
```

(b) LUBM Q3

```
SELECT ?x  WHERE {
 ?x rdf:type ub:Person .
 ?x ub:memberOf
   http://www.Department0.University0.edu
}
```

(c) LUBM Q5

图 6.1 LUBM Q1、Q3、Q5

上述三条查询在测试数据大小不断增长的情况下，其查询响应效率没有显著的降低，查询时间曲线相对平缓。如图 6.2 展示了这三条查询的增长曲线。

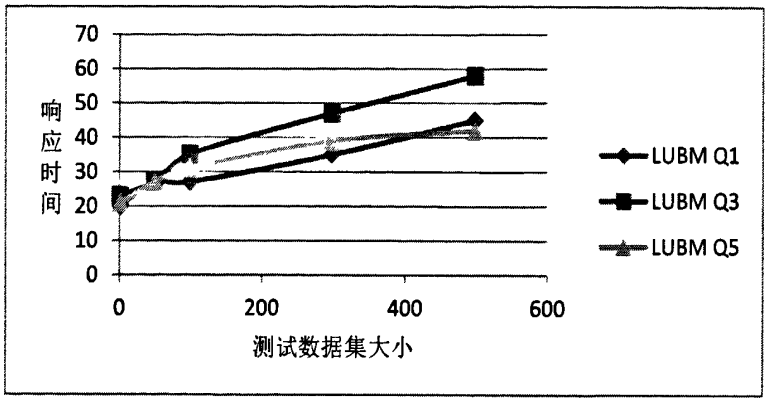


图 6.2 LUBM Q1、Q3、Q5 增长曲线

图 6.2 中的都是在 Map 阶段进行过 BloomFilter 过滤的结果，图 6.3 展示了未在 Map 阶段提前过滤的对比结果。随着测试数据的增大，在 Map 阶段提前过滤的效果越明显。

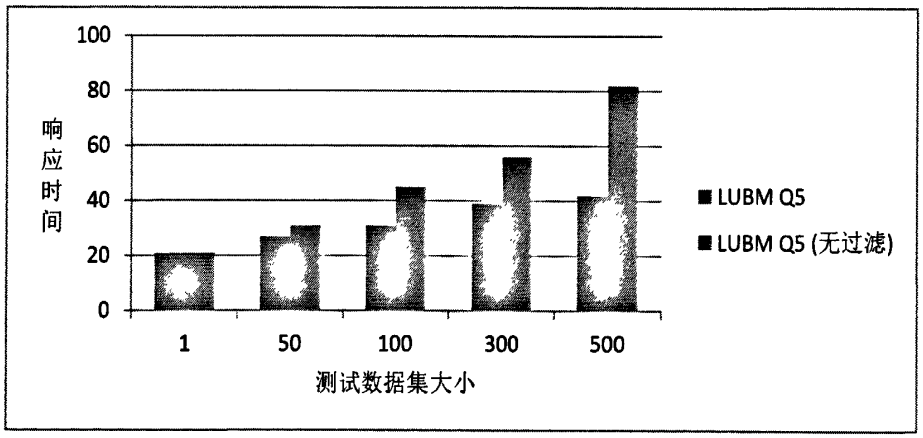


图 6.3 LUBM Q5 查询结果对比

6.2.2 LUBM Q2 查询结果分析

如图 6.4 所示，LUBM Q2 的响应时间增长相对较快。根据当前对用于创建 BloomFilter 的选择，LUBM Q2 中的各项 Triple Pattern 子句均不满足要求。因此在处理 LUBM Q2 时未进行任何过滤操作。所以 LUBM Q2 查询处理未能从本文描述的方法中获得性能上的改进。然而观察 LUBM Q2 响应时间的增长与测试数据的增长，在测试数据增长了 500 倍的同时，查询响应时间只增长了约 8 倍。因

此从这里也可以看出分布式系统在此时所展现的良好的可伸缩性。

```
SELECT ?x ?y ?z WHERE {
 ?x rdf:type ub:GraduateStudent .      // 1
 ?y rdf:type ub:University .           // 2
 ?z rdf:type ub:Department .           // 3
 ?x ub:memberOf ?z .                   // 4
 ?z ub:subOrganizationOf ?y .          // 5
 ?x ub:undergraduateDegreeFrom ?y // 6
}
```

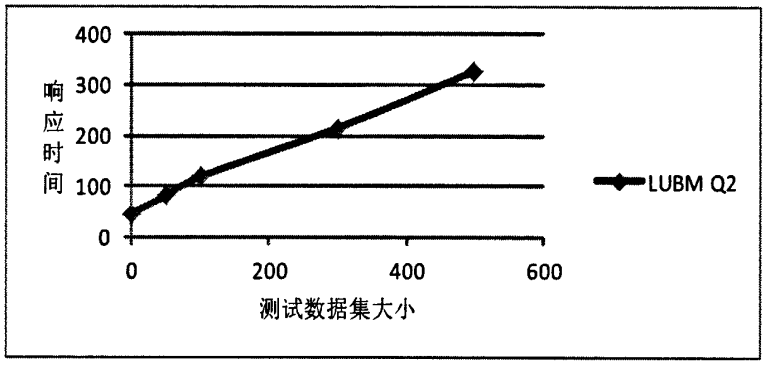


图 6.4 LUBM Q2 增长曲线

6.2.3 LUBM Q4 查询结果分析

如图 6.5 所示，LUBM Q4 查询与 LUBM Q1 等也相似，不同之处在于 LUBM Q4 中包含了更多的 Triple Pattern 子句。因此在查询处理上需要更多时间。

```
SELECT ?x WHERE {
 ?x rdf:type ub:Professor .
 ?x ub:worksFor
    <http://www.Department0.University0.edu> .
 ?x ub:name ?y1 .
 ?x ub:emailAddress ?y2 .
 ?x ub:telephone ?y3
}
```

图 6.5 LUBM Q4

6.2.4 LUBM Q6 查询结果分析

如图 6.6 所示，LUBM Q6 只包含了一个 Triple Pattern 子句。其中并不存在任何连接操作，因此实际上并不需要创建 MapReduce 任务来处理该连接。这里我们

依然测试了用 MapReduce 处理其的响应时间，主要是为了观测大量数据从 HBase 中读取后再经历从 Mapper 到 Reducer 的传递，以及 Reducer 写入 HDFS 的效率。图 6.6 中的横轴表示的是 LUBM 数据集 University 的数目，以及在该数据集下 MapReduce 任务中 Mapper 输出的字节数。

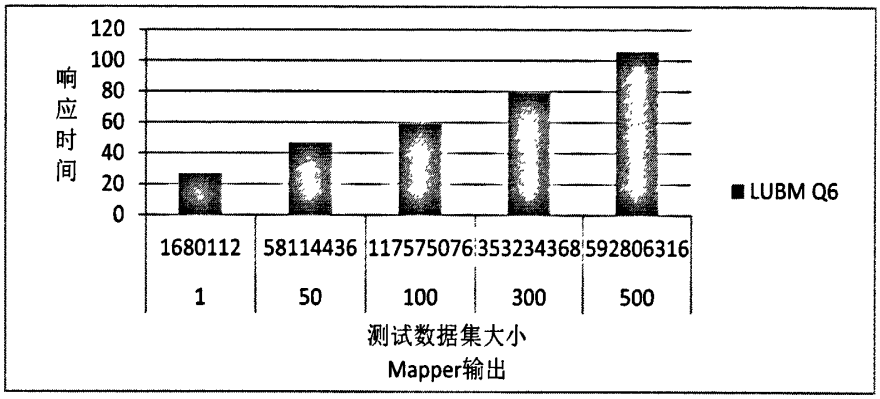


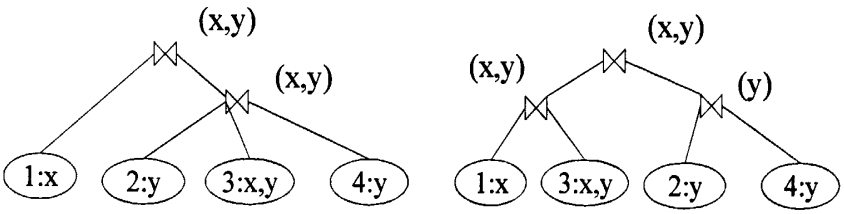
图 6.6 LUBM Q6 增长曲线

6.2.5 LUBM Q7 查询结果分析

图 6.7 展示了 LUBM Q7 以及两种不同的查询计划方案。

```
SELECT ?x ?y ?z WHERE {
 ?x rdf:type ub:Student .      // 1
 ?y rdf:type ub:Course .       // 2
 ?x ub:takesCourse ?y .        // 3
 <http://www.Department0.University0.edu/AssociateProfessor0>
    ub:teacherOf ?y //4
}
```

(a) LUBM Q7



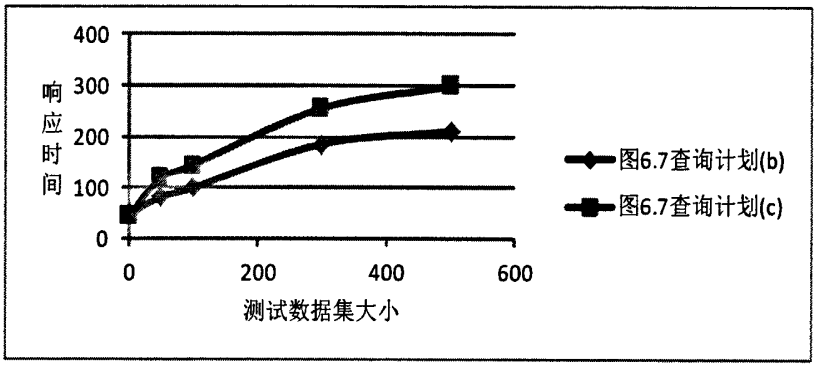
(b) 选用的查询计划　　(c) 未选用的查询计划

图 6.7 LUBM Q7 查询计划

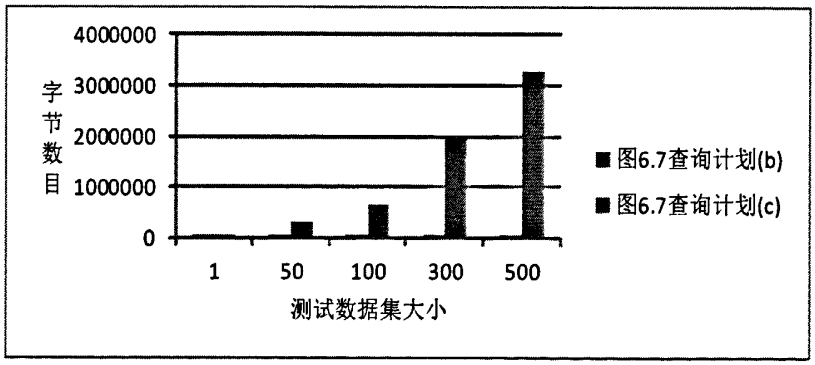
图 6.8 展示了两个查询计划对应的响应时间，以及 MapReduce 任务 Mapper

的输出比较。在查询响应时间上，查询计划（b）对应的处理方案可以在更短的时间内完成。在 Mapper 输出大小比较上，查询计划（b）对应的输出同查询计划（c）对应的输出相比几乎可以忽略不计。从输入输出上来看，查询计划（b）的响应时间应当能快很多，然而实际测试的结果表明两者间的差距没有那么巨大。

经过对 MapReduce 任务的分析，当前两个查询计划处理过程中的主要瓶颈在于 HBase 数据的读取。在当前 HBase 集群的配置下，HBase 只被分配了 1GB 的内存，这样在进行大量数据读取时候，HRegionServer 受限于内存大小不能高效的完成查询应答。如若 HBase 可以分配到更多的内存，那么数据读取上将会高效很多，那样的情况下更能评判出两个查询计划的优劣。



（a）查询响应增长曲线



（b）Mapper输出大小比较

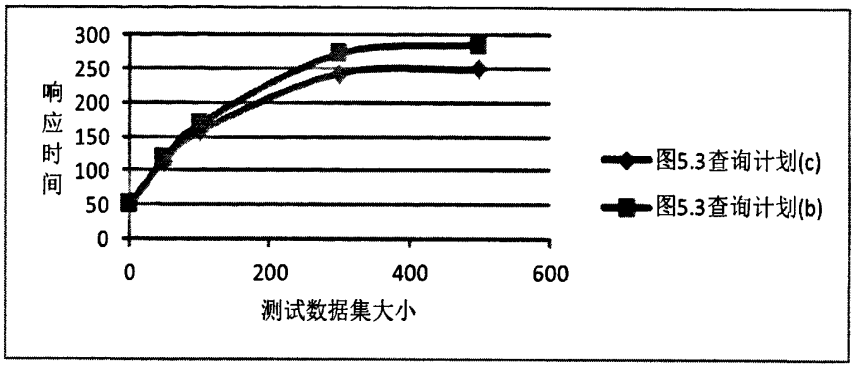
图 6.8 LUBM Q7 查询结果对比

6.2.6 LUBM Q8 查询结果分析

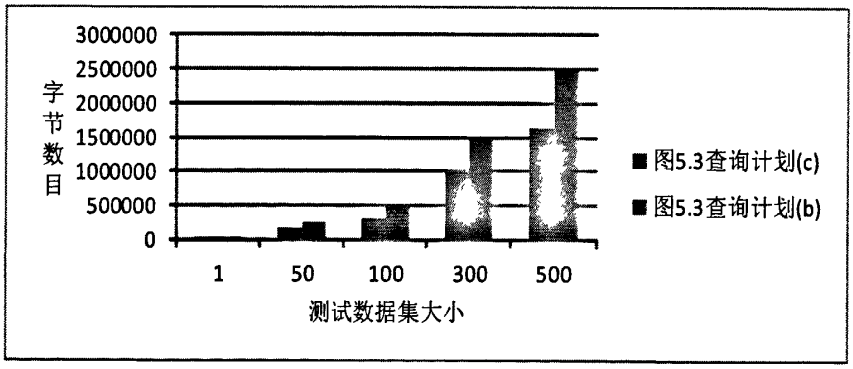
图 5.3 中已经展示过 LUBM Q8 表达式，以及针对 LUBM Q8 的两个不同查

询计划。图 6.9 展示了图 5.3 中两个不同查询计划对应的查询响应时间，以及 MapReduce 任务 Mapper 的输出比较结果。

图 6.9 所示的结果同 LUBM Q7 的结果类似，虽然选定的查询计划显著的减少了 MapReduce 任务的输入输出大小，但是整体查询性能的提升没有获得与之相对应的大量增长。当前系统的重要瓶颈在于 HBase 数据读取部分消耗了大部分时间，因此最终的查询性能提升未那么明显。



（a）查询响应增长曲线



（b）Mapper输出大小比较

图 6.9 LUBM Q8 查询结果对比

6.2.7 实验结果综合分析

表 6.3 展示了八条 LUBM 查询在不同数据集下的查询响应时间，从表中的数据可以看出，当前系统基本上能够支持大规模 RDF 数据集的存储与查询。即便是针对拥有复杂连接的 LUBM Q2 查询，系统依然可以在较短的时间内完成查询响应。从各个查询单独的分析结果可以看出当前我们采用的 MapReduce 任务处理方

案有效的降低了 MapReduce 任务中的 I/O 操作，并使的整体查询性能有了一定程度的提升。

然而从 LUBM Q7、Q8 不同查询结果的比较来看，当前获得的查询性能提升同降低了的 I/O 操作间并不完全匹配。经过对各个查询语句以及 MapReduce 任务处理过程中的日志分析，当前系统的一个瓶颈在于 HBase 读取操作占据了查询处理的相当一部分时间。根据 HBase 的特性以及当前 HBase 集群的配置状况来看，造成这一情况的主要原因在于当前 HBase 集群的内存不足。HRegionServer 只分配到 1GB 的内存，而 HBase 对内存大小相当敏感。从 HBase 官方文档来看，为了尽可能提高 HBase 的处理效率，需要给 HBase 节点分配足够的内存。在 HBase 节点内存充足的情况下，本文提出的 RDF 查询处理方案优势才会更明显。

6.3 本章小结

本章主要介绍了针对本文提出的 RDF 查询处理方案进行的实验情况。本章首先介绍了实验环境以及实验数据选择，随后给出了在不同 LUBM 测试数据集下的 LUBM 查询响应状况，接着逐条分析了各条查询结果，并对比了不同查询计划的查询响应结果，最终对实验结果进行了整体分析。

第7章 总结与展望

本文对当前 RDF 存储系统研究现状进行了深入分析后，提出了采用分布式存储系统 HBase 存储 RDF 数据，采用 MapReduce 并行计算框架处理 SPARQL 中 BGP 连接操作的整体方案。本文的工作为 RDF 存储系统的发展提供了一种新的合理选择。

7.1 特点与创新

本文提出的基于 HBase 的 RDF 存储系统方案有如下的特点与创新：

首先，我们提出采用 HBase 存储 RDF 数据。分布式 RDF 存储系统通常都需要面对数据划分策略的选择，但在我们的方案中 RDF 数据的分布依托于 HBase 自身提供的数据划分特性，从而避免陷入复杂的分布式环境细节处理当中。

其次，我们提出了 HBase 上存储 RDF 数据的具体表结构定义。在我们提出的方案当中，RDF 数据存储在 HBase 中 SPO、POS、OSP 三张数据表当中，充分利用了 HBase 提供的默认索引机制来对 RDF 数据进行索引。本文描述的方案在保证 RDF 查询性能的同时有效地减少了 RDF 数据的存储开销。

再者，在当前已有的 MapReduce 多路连接方法基础之上，我们进一步提出了 MapReduce 多路连接查询计划的生成决策方法。MapReduce 任务在运行处理过程中会涉及大量的 I/O 操作，因此我们在生成查询计划时在保证 MapReduce 任务数目一定的前提下尽可能考虑减少每个任务的 I/O 开销。我们结合 SPARQL 中 Triple Pattern 子句的特性以及查询中间结果的多寡来选取数据集在 Map 阶段进行过滤操作。本文提出的方案在保证 MapReduce 任务数目一定的条件下有效地减少了查询处理过程中的 I/O 开销。

最后，我们结合提出的 RDF 存储与查询方案创建了原型系统并采用 LUBM 测试数据集对本文提出的 RDF 查询处理方案进行了实验验证。

7.2 不足与缺陷

本文提出的方案当前也存在着一些不足与局限。

本文当前并未考虑HBase相关诸多参数的配置。譬如HBase数据是以Region形式交由多个HRegionServer进行管理的，因此我们需要考虑合理配置Region的大小，决定Region大小超过多少时才进行分裂。这对如(?S，P，?O)类型的查询性能极大相关，当前RDF数据存放在SPO、POS、OSP三张数据表中，(?S，P，?O)查询可能需要访问当量数据行，因此合理的切分Region是将会影响到该查询分布化的效果。

本文未涉及MapReduce任务的调优工作。MapReduce框架提供了大量的可供配置的参数，这些参数的配置对MapReduce任务的性能有重大影响。当前的测试工作只是利用了测试集群上固有的配置。

此外原型系统中的RDF查询处理功能当前只包含了SPARQL BGP连接处理。SPARQL包含的诸多特性需要得到进一步支持。

7.3 展望

在后续的研究工作当中，我们将努力解决当前方案中的各项不足与缺陷。我们将尝试对HBase集群以及MapReduce相关的各项参数进行配置测试，以在一定的硬件条件下获得尽可能好的查询响应性能。我们也将进一步研究RDF查询处理方法，以期将当前的查询计划方法进一步优化。此外我们也将考虑进一步支持更多的SPARQL algebra，以提供一个可以应用在生产环境中的RDF存储系统。

7.4 本章小结

本章总结了本文研究工作的创新点，然后分析了但前研究的不足之处，最后对后续工作进行了展望。

参考文献

[1] L. Moss, M. Brodie. Data Rich, but Information Poor. http://www.dmreview.com/issues/20020701/5341-1.html, 2002.

[2] N. Shadblt, W. Hall, and T. Berners-Lee. The Semantic Web Revisited[J]. IEEE Intelligent System, 2006(21):96~101.

[3] RDF Primer. http://www.w3.org/TR/REC-rdf-syntax

[4] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google File System[C]. In Proceedings of the 19th ACM Symposium on Operating System Principles, 2003.

[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[C]. In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06), 2006.

[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters[C]. In Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04), 2004.

[7] Hadoop. http://hadoop.apache.org

[8] HBase. http://hbase.apache.org

[9] C. Weiss, P. Karras and A. Bernstein. Hexastore: sextuple indexing for semantic web data management[C]. In Proceedings of the 34th International Conference on Very Large Data Bases (VLDB'08), 2008.

[10] Y. Guo, Z. Pan, and J. Heflin. LUBM: a benchmark for OWL knowledge base systems[J]. Web Semantics: Science, Services and Agents on the World Wide Web, 2005(3): 158~182.

[11] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query

[12] ARQ. http://jena.sourceforge.net/ARQ

[13] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2[C]. In Proceedings of the 1st International Conference on Semantic Web and Databases (SWDB'03), 2003.

[14] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning[C]. In Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07), 2007.

[15] C. Weiss, P. Karras and A. Bernstein. Hexastore: sextuple indexing for semantic web data management[C]. In Proceedings of the 34th International Conference on Very Large Data Bases (VLDB'08), 2008.

[16] M. Cai and M.R. Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network[C]. In Proceedings of the 13th International Conference on World Wide Web (WWW'04), 2004.

[17] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web[C]. In Proceedings of the 6th International Semantic Web Conference (ISWC'07), 2007.

[18] S. Harris, N. Lamb, and N. Shadbolt. 4store: The Design and Implementation of a Clustered RDF Store[C]. In Proceedings of the 8th International Semantic Web Conference (ISWC'09), 2009.

[19] A. Owens, A. Seaborne, and N. Gibbins. Clustered TDB: A Clustered Triple Store for Jena[C]. In Proceedings of the 18th International Conference on World Wide Web (WWW'09), 2009.

[20] M.F. Husain, P. Doshi, L. Khan and B. Thuraisingham. Storage and retrieval of large RDF graph using Hadoop and MapReduce[C]. In Proceedings of the 1st International Conference on Cloud Computing (CloudCom'09), 2009.

[21] J. Myung, J. Yeon, and S. G. Lee. SPARQL Basic Graph Pattern Processing with Iterative MapReduce[C]. In Proceedings of the Workshop on Massive Data Analytics on the Cloud (MDAC'10), 2010.

[22] Y. Tian, J. Du, H.F. Wang, Y. Ni, and Y. Yu. HadoopRDF: A Scalable RDF Data Analysis System. Submissions of Semantic Web Challenge, 2010.

[23] J. Broekstra and A. Kampman. Sesame: A generic Architecture of Storing and Querying RDF and RDF Schema[C]. In Proceedings of the 1st International Semantic Web Conference (ISWC'02), 2002.

[24] 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[C]. In Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM'09), 2009.

[25] D.J. Abadi. Column stores for wide and sparse data[C]. In Proceedings of the 3rd Biennial Conf. on Innovative Data Systems Research (CIDR'07), 2007.

[26] C. Lam. Hadoop in Action[M]. Manning Publications Company, 2010:121~122.

攻读硕士学位期间主要的研究成果

[1] Jianling Sun, Qiang Jin. Scalable RDF Store based on HBase and MapReduce. In Proceedings of the 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE'10), 2010.

致谢

两年半的研究生生活即将结束，在这期间师长、同学、家人、朋友都给予了我很大支持与帮助。

首先我要由衷地感谢我的导师孙建伶教授，孙老师严谨的治学研究态度让我受益匪浅。在整个研究生期间，我在实习工作与研究工作方面都得到了孙老师的大力支持与指导。在撰写毕业论文期间，从最初的开题到具体论文的书写，孙老师都在一直关注，并定期检查指导我论文的进展情况。论文整体架构与脉络的均得益于孙老师的指导意见。在孙老师指导下，我的全局观与系统性思维有了较大的提高。

其次我要感谢的是实验室的全体成员，特别要感谢李崇欣、杜啸飞、于猛、方加果、胡金栋、高晖、彭德跃，我在论文撰写与实验中遇到的大量问题均得到了他们及时耐心的帮助。

我还要感谢道富技术中心项目组的同伴，特别感谢我的项目经理王燕芬与项目组长陈一稀，感谢他们在平时工作中给我的信任与帮助，让我得以在项目工作中取得巨大的进步。

最后我要感谢我的家人，感谢一直以来对我的关心支持与理解。

金 强

2011年1月于求是园