硕士专业学位论文

基于压缩位图索引的 RDF 数据存储与管理

RDF Data Storage and Management Based on Compressed Bitmap Index

作者：

导师：

北京交通大学

2017 年 5 月

学位论文版权使用授权书

本学位论文作者完全了解北京交通大学有关保留、使用学位论文的规定。特授权北京交通大学可以将学位论文的全部或部分内容编入有关数据库进行检索，提供阅览服务，并采用影印、缩印或扫描等复制手段保存、汇编以供查阅和借阅。同意学校向国家有关部门或机构送交论文的复印件和磁盘。学校可以为存在馆际合作关系的兄弟高校用户提供文献传递服务和交换服务。

（保密的学位论文在解密后适用本授权说明）

学位论文作者签名：　　导师签名：

签字日期：　　年　　月　　日　　签字日期：　　年　　月　　日

学校代码：10004　　密级：公开

北京交通大学

硕士专业学位论文

基于压缩位图索引的 RDF 数据存储与管理

RDF Data Storage and Management Based on Compressed Bitmap Index

作者姓名：　　学　　号：15125012

导师姓名：　　职　　称：

工程硕士专业领域：电子与通信工程　　学位级别：硕士

北京交通大学

2017 年 5 月

致谢

摘要

随着资源描述框架（Resource Description Framework，RDF）在各个领域的广泛应用，如何对海量 RDF 数据的存储与管理成为近年来的研究热点。现有的 RDF 数据管理系统大都采用传统的关系型数据库来存储数据，这种方式已难以高效地管理海量数据。如何设计一种高性能、可扩展为分布式的的 RDF 数据存储和管理系统具有重要意义。本文设计了一种基于位图索引的 RDF 数据存储方案，并实现了基于该存储方案的 RDF 管理系统，最后通过系统测试验证了该方案的可行性与有效性。

本文研究工作主要包括以下几个方面。

(1) 总结了现有的 RDF 数据存储方案。分析了当前主流的数据存储技术及 RDF 数据存储模型的优缺点，并对其进行了简单的分析与总结。

(2) 提出了一种基于位图索引的高扩展性底层存储方案。该方案在持久层将 RDF 数据文件分块进行顺序存储，实现了系统的可扩展性；同时为 RDF 关键词构建基于压缩位图的查询索引，降低了运行时内存资源消耗。

(3) 设计了基于本方案的数据查询算法。该算法能够充分利用位图索引逻辑计算的性能优势，保证了高效的查询效率。

(4) 实现了基于本方案的 RDF 数据存储和查询系统 fishdb，并采用测试数据集在单机伪分布式系统环境下对该系统进行了性能测试。与开源 RDF 管理系统 Google Cayley 的相比，fishdb 能够以较小的内存资源消耗为代价换取较高的查询性能提升，验证了本方案的可行性和有效性。

关键词：RDF；数据存储；查询；位图索引；压缩

ABSTRACT

With the popular usage of Resource Description Framework (RDF) in various fields, how to store and manage the massive RDF data becomes a research focus in recent years. Most of the existing RDF data management systems use the traditional relational database to store data, which is difficult to manage massive data efficiently. Therefore, how to design a high-performance, scalable and distributed RDF data storage and management system is of great significance. In this paper, we designed a RDF data storage scheme based on the bitmap index, then implemented the RDF management system based on this scheme.

The main work of this paper is as follows.

(1) Summarized the existing RDF data storage schemes. This paper analyzed the advantages and disadvantages of the current mainstream data storage technology and RDF data storage model, and made a simple analysis and summary.

(2) Proposed a high scalable underlying storage scheme based on the bitmap index. The RDF data are split into blocks which stored orderly in the persistence layer to achieve the scalability of the system. RDF terms are constructed on the compressed bitmap, which reduces the memory consumption during operation.

(3) Designed the data query algorithm based on this scheme, which can make full use of the performance advantage of the bitmap index logic operation and ensure the query efficiency.

(4) Implemented the RDF data storage and query system, which named *fishdb*, based on this scheme, and tested the performance of this system in a single machine pesudo distributed environment. Compared with the open source RDF management system *Google Cayley*, the *fishdb* can be used for higher query performance at the expense of smaller memory resource consumption, which verifies the feasibility and effectiveness of the proposed scheme.

KEYWORDS： RDF; data storage; query; bitmap index; compression

目录

1 引言

1.1 研究背景及意义

万维网的产生和发展，构建起自人类文明诞生以来最大的信息资源库。这个庞大的信息资源库囊括了几乎任何知识领域中的文档和媒体资源。通过网际互联和搜索引擎，这些信息资源能够通过被瞬时的检索和获取。随着互联网技术和产业规模的迅速发展，信息资源的规模和数量呈现出指数级的膨胀态势，进入了海量数据时代。与之而来的是，如何从这些海量的信息中获取最符合搜索目的信息变得越来越困难。当前比较通用且技术相对成熟的搜索引擎，属于全文搜索引擎（如 Google，Bing，Baidu），这些搜索引擎能够非常高效的完成基于关键词（term）的查询任务，比如，查询关于阿里巴巴的信息，可以通过关键词“阿里巴巴”进行搜索。但如果涉及到一些语义相关任务时，这些全文搜索引擎表现的远远不能满足用户需求，比如查询“阿里巴巴现任首席执行官曾经在那些公司工作过”。用户希望得到更为精准的信息回答，而不仅仅是提供一大堆基于关键词索引的网页。在这一问题的背后，是关于 web 上的资源信息如何能够被计算机所理解和处理的问题。

面对如何能够让计算机理解和处理资源信息这一问题，Tim Bernes-Lee 提出了语义网[1]（Semantic Web）的概念。通俗来说，语义网就是通过给万维网上的数据资源添加一些可以被计算机所能够“理解”的语义数据信息，使得万维网的数据资能够被计算机理解，甚至可以推理，从而可以处理复杂的用户查询请求。语义网并非是独立于万维网之外另一种类型的网络，它是对万维网的的扩展。它在万维网的数据之上，添加一层富含语义信息的网络。在语义网中语义信息数据被赋予了明确界定的意义，从而能更好的使人与计算机沟通与协作。

当前，语义网领域存在多种描述资源和关系的协议和框架。其中，资源描述框架[2]（Resource Description Framework，RDF）是相对而言比较成熟的一套协议框架，也是是万维网联盟所倡导和推广的元数据管理方案。RDF 采用三元组（s，p，o）的模式对资源及其关系进行描述。其中，s 表示主语（subject），p 表示谓语（predicate），o 表示宾语（object）。例如，“Baidu”的首席执行官是“Robin Li”这条信息，可以使用三元组（“Baidu”，<http://company/CEO>，“Robin Li”）来表示。其中，<http://company/CEO>是谓语，表示“首席执行官”这条语义，使用统一资源标识符（Uniform Resource Identifier，URI）进行标识；“Robin Li”是宾语，是“Baidu” 在<http://company/CEO>属性上的值。

含有语义层面的相关研究和实践已经有 30 多年的历史，这些研究和实践集中于信息的集成，搜索，分析乃至推理等方面。伴随着万维网技术的不断演进，及相关产业的蓬勃发展，网络的规模不断扩张，数据增长的速度也飞速发展，数据资源的语义信息也越发丰富。RDF 由于其自身的的简洁性和灵活性，被越来越多的应用于各个领域来管理信息资源，如社交网络，搜索引擎，生物技术等。著名的搜索引擎公司 Google 和社交网络公司 Facebook 等都于早些年开始采用 RDF 来构建语义网络，对资源的元数据进行管理。RDF 的广泛应用使得 RDF 数据的规模不断膨胀。据统计，到 2012 年 3 月份，仅 W3C 的开放链接数据[3]（LinkingOpenData）项目中发布的 RDF 数据量就高达 520 亿多个三元组，这个数据是 2011 年 9 月份 310 亿的 1.7 倍多，是 2010 年 5 月份 64 亿的 8 倍多[31]。不同的应用场景面对不同规模的数据。如何对 RDF 数据，尤其是海量的 RDF 数据进行高效的存储和管理成为一个亟待解决的问题。学术界和工业界对 RDF 数据管理的很多方面进行了较为详细的研究及系统方案的设计，集中于解决 RDF 管理相关的问题，如存储模型，底层物理存储系统，查询的链接操作等方面。但目前仍然缺乏较为通用的 RDF 数据管理方案。

1.2 研究现状及热点

语义网的数据管理问题一直是语义网研究领域的热点。ISWC，WWW，SIGMOD 等国际会议均有研究成果发表。学术界和工业界对此都有愈发火热的研究趋势。

在学术界已经有相当多的 RDF 管理系统被设计和实现，比较著名的有 RDF-3X[4][23]，Jena[5][26]等。这些系统的一个特点是，采用传统的集中式关系型数据库来对 RDF 进行存储。RDF 数据被拆分或组织成相关的表结构，然后存储在关系型数据库中。在查询时，将面向 RDF 的查询语言转化为面向关系型数据库的查询语言，从而作用于关系型数据库上。由于当前关系型数据库技术比较成熟，因此该实现方案的系统稳定性比较强。但关系型数据库的存储模型与 RDF 数据模型的差异较大，因此在数据的存储转换和查询上开销比较大。工业界的研究里面，近年来由 Google 开源的 Cayley 更为激进，它直接提供了面向异构的查询引擎，后端的数据存储设施可以自由选择，如关系型数据库，非关系型数据等。Cayley 的这种设计方案实现了存储和查询的分离，给存储上带来了很大的灵活性，可以根据 RDF 数据特点选择或者实现适合 RDF 数据模型的解决方案。但这样的同时带来了无法根据后端存储特点来定制查询引擎以提高查询性能的问题，这是为换取通用性而引入的代价。

单机系统受限于系统资源的扩展性，无法满足数据规模增长带来的存储和查询需求。因此，面向分布式的 RDF 数据管理系统是解决海量 RDF 数据管理难题的关键[28][29][30]。同时也是这方面的研究是近两年的研究热点，但这一方面还处于研究的初始阶段，目前还没有成熟的系统方案出现。

1.3 本文研究内容

随着 RDF 在各个领域的广泛应用以及数据量的急速增长，如何高效地管理海量数据是近年来的研究热点。现有的数据管理系统大都采用传统的关系型数据库来存储数据，这种方式已难以高效地管理海量数据。如何设计一种高效且可扩展的 RDF 数据存储和查询系统来解决海量数据管理问题具有重要意义。本文结合基于位图的索引引擎以及分布式的底层存储模型，提出一种基于位图索引的 RDF 数据管理方案，并设计和实现了基于此方案的查询算法。同时，将本毕设的实现方案与现有其他方案进行对比，验证了本实现方案的可行性。

本毕设的研究内容及工作包括以下几个方面。

(1) 总结当前 RDF 数据管理方面的解决方案及相关技术。相关技术总结了当前比较主流且成熟的单机存储技术和分布式存储技术，以及基于这些存储技术的 RDF 存储管理方案。

(2) 提出了一种基于可扩展为分布式的数据存储模型。将 RDF 行数据文件按照块（chunk）存储，将每个块利用元数据索引进行组织和管理，最大程度的降低了硬盘存储开销，并提高了数据的容错性。

(3) 利用实现该存储模型，基于压缩的位图索引实现了的查询算法。该算法能够充分利用位图索引逻辑计算的优势，保证了高效的查询性能基础；同时，使用可压缩的数据结构，降低了运行时内存资源的开销。

(4) 采用测试数据集在单机伪分布式系统下对存储模型与查询算法进行了实验评估，验证了存储模型和查询算法的可行性，并且与 Google Cayley 进行对比，通过对比实验证明了本文提出的方案的有效性。

1.4 本文组织结构

根据上述研究内容，本文共有六章，论文其余部分章节组织如下。

第二章介绍了语义网的相关技术，对语义网几种关键技术如 RDF、SPARQL 标准查询语言进行了介绍。随后对单机存储技术和分布式存储技术进行了简要介绍。最后总结了现有的数据存储模型，分析了现有的关系型数据存储几种普遍的

存储模式的优缺点，介绍了一部分基于分布式计算平台的 RDF 数据存储与查询方法，并对其进行了简单的总结。

第三章介绍了基于分布式文件系统的底层存储模型及位图索引结构，并介绍了该存储方案的实现细节。

第四章介绍了基于压缩位图索引的数据查询算法，及其查询方案的实现细节。

第五章对本文提出的存储模型与查询算法进行了性能测试，并对性能测试结果进行了分析。

第六章为回顾了本文的主要工作，对研究内容和成果进行了简要的总结，并提出了今后需要进一步进行研究的内容和方向。

2 相关技术综述

本章将简要介绍 RDF 及数据存储系统涉及到的一些协议框架和底层存储技术方案。协议框架主要包括语义网涉及到的相关概念和技术，底层存储技术方案主要围绕单机存储技术和分布式存储技术展开。并且讨论了现有的基于这些技术的 RDF 存储方案，最后对本章进行了小结。

2.1 语义网相关技术

本节将简要介绍语义网相关的技术和协议，包括资源描述框架 RDF 和基于该协议的查询语言 SPARQL。

2.1.1 RDF

资源描述框架（Resource Description Framework，RDF）是描述结构信息的一种形式化语言。RDF 的目标是让应用程序能够在 Web 上交互数据，并且仍然保留它们原有的含义。与一般的文本标记语言不同，这里的主要意图不再是正确地显示文本，而是允许进一步处理和重新组合其中包含的信息。所以，RDF 经常被看作是发展语义万维网的基本表示形式。

RDF 的发展始于 20 世纪 90 年代，第一个官方的规范是由 W3C 在 1999 年发表的，当时 RDF 面向的用途主要是对万维网资源的元（meta）数据表示。元数据指描述数据集或文档信息的数据。在 1999 年，这些文档信息主要是指网页资源，RDF 能够帮助说明这些网页的作者或着版权信息。后来语义万维网的构想被扩展到了一般语义信息的表示，超越了简单的 RDF 数据以及作为其主要描述对象的 Web 文档，这是 2004 年发表一个被重建和扩展的 RDF 规范和动机。

(1) RDF 的数据模式：三元组

RDF 采用三元组（s，p，o）的模式对资源及关系进行描述。其中，s 表示主语（subject），p 表示谓语（predicate），o 表示宾语（object）。例如，“Baidu”的首席执行官是“Robin Li”这条信息，可以使用三元组(“Baidu”,<http://company/CEO>，“Robin Li”）来表示。其中，<http://company/CEO>是谓语，表示 CEO 这条语义，使用统一资源标识符（Uniform Resource Identifier，URI）进行标识；"Robin Li"是宾语，是“Baidu”在<http://company/CEO>属性上的值。

(2) RDF 的语义模式：图

一个 RDF 文档描述了一个有向图，即一组由有向边（“箭头”）连接起来的结点。结点和边都用标识符进行标记以便区分。图 2.1 展示了一个简单的图，它包含两个结点和一条边。

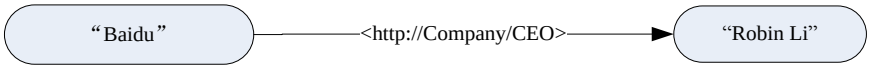


图 2-1 RDF 实例图

Figure 2-1 RDF instance diagram

RDF 被设计成一个图结构的重要的原因是，RDF 并不是为了结构化文档而设计的，而是为了描述感兴趣的对象（在 RDF 中通常叫做“资源”）之间的一般关系而设计的。如图 2-1 所示的 RDF 图，如果把其中的标记翻译成指代的那些对象，那么这个 RDF 图就可以被用于表达这样一个含义：“Baidu”公司的首席执行官是“Robin Li”。在这种情况下，公司和首席执行官之间的关系显然在任何意义下都不是层次上的从属于任何一个资源。所以 RDF 把这种关系作为构建信息的基石。许多这样的关系在一起自然构成了图，而不是层次化的树结构。

选择图的另一个理由是，RDF 是为万维网和其他电子网格上的数据设计的一种描述语言。在这些环境中的信息通常被分散地存储和管理，而合并不同来源的 RDF 数据也确实是很容易的。一般允许图含有多个互不相连的部分，即各个部分之间没有任何边的子图。所以，RDF 图更适合分布式信息源的组合。

2.1.2 SPARQL

在 RDF 模型诞生很久以后，才出现了查询 RDF 数据的标准化方式。已在使用的专用查询语言有 6 种或更多，每种语言都有着自己的独特性和不兼容性。这种情形不利于对不同的 RDF 存储系统使用通用查询，使可移植数据的愿景大打折扣。幸运的是，W3C 在 2008 年发布了 SPARQL 协议和 RDF 查询语言[7]（SPARQL Protocol and RDF Query Language，SPARQL）。SPARQL 采用了类似广泛应用于关系型数据库的 SQL 协议，表述方式简洁明了。

SPARQL 查询的最简单形式尝试匹配图形的各部分，并选择一个或多个表达为图形模式的重要位置的变量。此方法类似于 SQL SELECT 查询方式，但它在图形上而不是在表上操作，它的基本格式如下：

```
SELECT variable-list WHERE
{
 graph pattern
}
```

其中，variable-list 代表要查询的变量的列表，graph pattern 是这些变量要满足的图模式（graph pattern），即查询变量要满足的约束。

图模式是使用图形表达一种结构关系，引用了节点和链接节点的圆弧。在上节中提到，在 RDF 中节点转换为主语实体，连接到属性的圆弧将它们连接到图形中的其他节点。如果想要询问特定节点的问题，可以在模式的主语位置指定这些节点。如果想要知道一个特定属性的值，可以在模式的谓语位置指定该属性。任何不想指定的元素都可由一个变量来表示，该变量将映射到该位置存在的任何可能值。如果不指定模式的任何部分，则会实际要求图形中的所有元组扁平化到一个结果集中。结果集的内容取决于选择了哪些变量。

假设对数据运行一个主语查询，而且知道所描述的实体的身份。如果想要询问一个特定主语的信息，可以指定图形模式中的该值，并选择与该主语关联的所有谓语-宾语对。简单来讲可以说，“告诉我您知道某个特定资源的所有信息”。比如，要找到 commlab 实验室中学生实体 s_1 资源的更多信息，SPARQL 查询将类似于：

```
SELECT ?p ?o WHERE
{
  <https://commlab/student/s_1> ?pred ?obj .
}
```

结果将类似于表 2-1 所示。

表 2-1 SPARQL 查询结果

Table 2-1 SPARQL query result

Pred	obj
<http://commlab/person.name>	“wee”
<http://commlab/person.age >	24
<http://commlab/study.type>	“master”
<http://commlab/study.follow>	“Dr.Mike”

2.2 单机存储技术

单机存储引擎是哈希表，B+树等数据结构在机械硬盘，固态硬盘（SSD）等持久化介质上的实现。单机存储系统的理论来源于关系型数据库。单机存储系统是单机存储引擎的一种封装，对外提供文件，键值，表格等数据模型的接口。单机存储系统的理论来源于关系型数据库。单机存储技术已经有很多年的理论和实践经验，技术非常成熟，是许多 RDF 数据管理方面的优先选择方案，尤其是在数

据量较小的情况下。而存储引擎是存储系统的发动机，直接决定了存储系统的性能及功能，本节将对单机存储引擎的核心技术进行简要介绍。

2.2.1 哈希存储引擎

基于哈希表结构的键值模型存储系统，一般仅支持随机读取，不支持顺序扫描。这里，以最具代表性的 Bitcask 存储模型为例，它是一种典型的日志型键值模型。所谓日志型，是指其持久层不支持随机写入，而是像日志一样仅支持追加操作。索引数据则是以哈希表的结构存在于内存中。

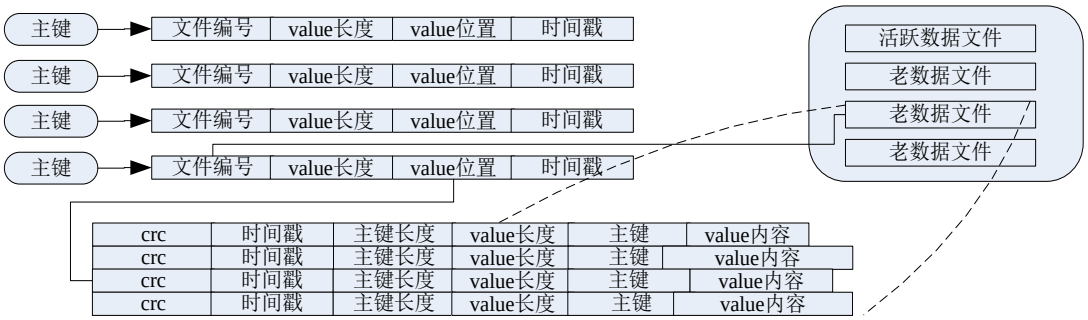


图 2-2 Bitcast 数据存储模型

Figure 2-2 Bitcast Data storage model

如图 2-2 所示，Bitcast 的内存中索引数据以哈希表的形式存在，其中主键映射的索引值是由文件编号，value 长度，value 位置和时间戳组成。通过文件编号，value 的位置和长度，引擎能够定位到 value 在硬盘中的位置，然后发起数据访问，读取指定长度的数据，即可获取目标值。一般而言，一条数据对应的索引值的数据长度要远远小于值本身的数据长度。因此，发起一次读取请求，相当于一次内存中的哈希查找和一次硬盘读取，时间复杂度 O（1）。同理，当数据写入时，首先会写入硬盘的数据（日志）文件，然后更新内存中哈希索引。

值得注意的是，任意时刻系统中只有一个文件支持写入，称为活跃数据文件（active data file），其余的称为老数据文件（older data file），只支持只读操作。并且要定期对文件进行合并操作，去掉冗余数据。为了能在宕机之后快速恢复，可以将索引数据持久化到线索文件（hint file）中。

通过以上可以看到，Bitcask 引擎通过哈希存储索引能够支撑快速的读取操作，同时日志型文件存储能够将随机写入转化为顺序写入，提高了写操作的吞吐量。

2.2.2 B+树存储引擎

B+树是一种树数据结构，是一个 n 叉树，每个节点通常有多个子节点，一棵 B+树包含根节点、内部节点和叶子节点。根节点可能是一个叶子节点，也可能是一个包含两个或两个以上孩子节点的节点。相比哈希存储引擎，B+树存储引擎不仅支持随机读取，还支持范围扫描。在 MySQL[8] 的 InnoDB 引擎中，有一种称为聚集索引的特殊索引，行的数据存在其中，组织成 B+树的数据结构。

如图 2-3 所示，MySQL InnoDB 按照页面（Page）来组织数据，每个页面对应 B+树的一个节点。其中，叶子节点保存每行的完整数据，非叶子结点保存索引信息。数据在每个节点中有序存储。进行数据查询时，需要从根节点开始二分查找直到叶子节点。每次读取一个节点，如果对应的页面不在内存中，需要从硬盘中读取并缓存起来，其中，B+树的根节点是常驻内存的。因此，对 B+树进行一次查询，最多需要进行（h-1）次磁盘 IO，复杂度为 O（h）=O（$\log_d N$），其中，h 为 B+树的高度，d 为非叶子结点出度，N 为存储元素个数。进行写入修改操作时，首先需要提交修改日志，然后更新内存的 B+树。如果内存中被修改的页面超过一定比率，后台线程会将这些页面刷到磁盘进行持久化。

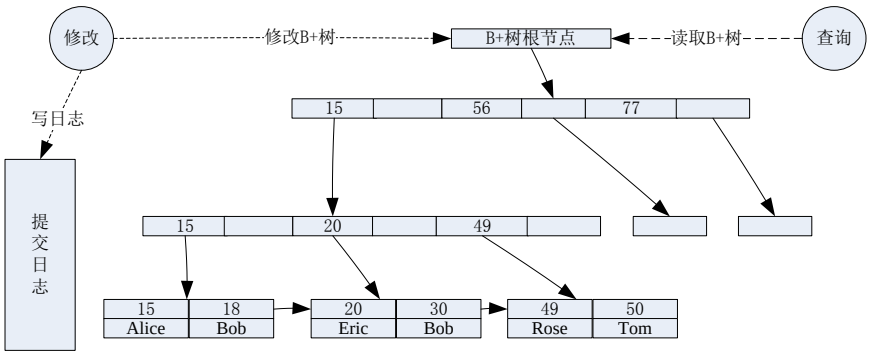


图 2-3 B+树数据存储模型

Figure 2-3 B+Tree Data storage model

对 B+树存储引擎来说，页面的频繁的换入换出要求对缓冲区有较好的管理策略。常见算法有两种：LRU 算法和 LIRS 算法。LRU 算法淘汰最长时间没有读/写过的块。但 LRU 存在一个致命问题，如果一个查询需要进行全表扫描，则会将缓冲区中的大量页面被替换。现代数据库一般采用 LIRS 算法，将缓冲区分为两级，数据首先进入第一级，如果数据在较短时间内被访问多次，则成为热点数据进入第二级。每一级内部还是采用 LRU 替换算法，但避免了 LRU 在面对全表扫描时会出现污染缓冲区的问题。

2.2.3 LSM 树存储引擎

LSM 树[8]（Log Structured Merge Tree）的主要目标是快速建立索引。B+树是建立索引的通用技术。但是，在面对大量并发插入数据的情况下，B+树需要大量的硬盘随机 IO，对索引建立的速度造成非常大的影响。特别地对于那些索引数据量特别大的情况，这种影响尤其明显。面对这种对写入性能比较敏感的情况，LSM 树的思想非常朴素，就是将对数据的修改增量（包括索引）保持在内存中，当内存中的数据量到达一定阈值之后，将这些数据批量写入磁盘。通过这种方式将随机 IO 转换成顺序 IO。

LSM 树的主要思想是划分不同等级的树。以两级树为例，可以想象成一份索引数据由两个树组成，如图 2-4 所示，一棵树存在于内存中，另一棵树存在于硬盘中。内存中的树不一定是 B+树，也可能是其他树（如 AVL 树等）。因为数据量大小是不同的，没必要牺牲 CPU 来达到最小的树高度。而存在于磁盘的是一颗 B+树。

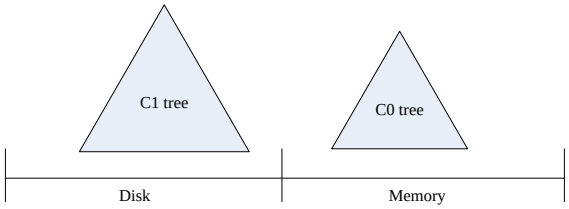


图 2-4 LSM 树结构

Figure 2-4 LSM Tree structure

数据会首先插入到内存中的树结构中。当内存中树结构元素个数超过一定规模后会进行合并操作，合并操作如图 2-5 所示。合并操作从左至右遍历内存中的叶子结点与硬盘中的叶子节点进行合并。当被合并的数据量达到磁盘的存储页的大小时，会将合并后的数据持久化到硬盘，同时更新父亲节点对叶子结点的指针。

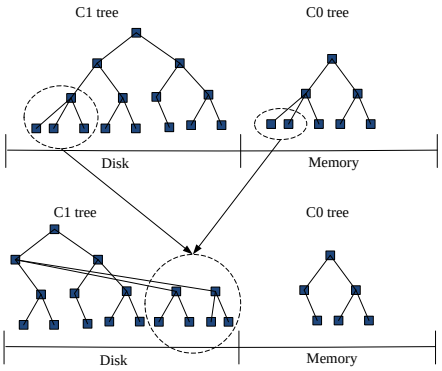


图 2-5 LSM 树合并示意图

Figure 2-5 LSM tree merging process diagram

之前存在于磁盘的被合并的叶子结点并不会被删除，这些数据会拷贝一份与

内存中数据一起顺序写入到磁盘。LSM 树会提供一些机制来延时回收这些磁盘空间。磁盘中的树的非叶子结点数据会在内存中缓存。当数据查找时，会首先对内存中的树进行查找，如果没有找到则继续对硬盘中的树进行查找。

LSM 树存在一个较为明显的问题。一旦数据量达到一定量级，硬盘中树的规模相应的变的非常大。导致的后果是写入数据时合并速度变慢（一定程度上退化到 B+树所面临的问题）。一个有效的解决方案时，在硬盘中建立不同层次（Level）的树，越低层次的树，规模越大。假设内存中树为 C_0，磁盘中的树按照层次分为 C_1，C_2，… C_K，则合并顺序为（C_0，C_1），（C_1，C_2），（C_2，C_3）…（C_{k-1}，C_k），缓解了合并速度的问题。Google 开源的高性能单机存储数据库 LevelDB 就是该思路的一个具体实现。

LSM 树通过先对内存进行修改，进而通过合并操作批量顺序写入硬盘，达到最优的写入性能。当然也损失了一部分读取性能。

2.2.4 位图存储引擎

位图索引[9]（bitmap index）技术是一类特殊的数据库索引技术，其索引使用比特（bit）数组（或称 bitmap、bitset、bitstring、bitvector）进行存储与计算操作。

(1) 位图索引的定义

下面给出位图索引的定义：位图索引可以看作是存储了大量 bit 位的 bit 序列，并且通过这些 bit 序列上的按位操作来响应查询请求，同时每个 bit 序列中的位数与数据表中的行数是一致的。

传统位图索引适用于低基数（cardinality）列。在索引技术中，列的基数描述了一个列中数据的散列程度，表示该列中不同值的个数；其中一个最极端的低基数的样例就是布尔类型，只含有 true 和 false 两种值，所以布尔类型列的基数值为 2。

Identifier	HasIphone	Bitmap(yes)
a	yes	1
b	yes	1
c	no	0
d	yes	1
e	no	0
f	no	0
g	yes	1
h	no	0

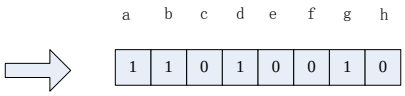


图 2-6 位图索引示意图

Figure 2-6 bitmap index diagram

图 2-6 是位图索引的一个直观示意图。其中，Identifier 列是每一行的唯一标识，

HasIphone 是索引列，其值域为{“yes”，“no”} ，这样通过一个比特就能表示其值。如图 2-6 所示，使用 1 表示“yes”，0 表示“no”,则该索引列可以表示成左侧表格的第三列。该列一共有 8 个比特，因此可以用右侧所示的一个字节来存储该位图索引。

传统位图索引由于其结构的特殊性，所以在存储空间和特定列的查询性能上都存在一定优势。以下两个关键因素使得位图索引成为查询密集型应用的利器：

1) 利用计算器硬件对位逻辑操作（AND，OR，XOR）的强有力支持，从而使单列内部的操作可以有效的转化为按位逻辑操作；
2) 多列之间的结果聚合也可以有效转化为按位逻辑操作。

但同时也存在一定的局限性：

1) 索引大小随着基数的数目增加而线性增加，从而导致高基数列索引查询性能较差；这也是传统位图索引的最大劣势所在；
2) 更新操作慢，由于更新操作的锁只能由单用户获取，并且需要同时锁住很多索引，故并发性较差；
3) 适用于只读，较少更新或者追加（append）数据集上的查询操作。

位图索引的适用场景：数据仓库，决策系统，图数据库。不适合事务性数据库。

(2) 位图索引的压缩

由于位图数据存在很大的稀疏性，逐渐发展出一系列基于位图的压缩技术。这里着重介绍两种最具代表性的压缩压缩算法：WAH[10]和 Roaring[11]。

WAH 算法的核心在于两点：字对齐（Word-Aligned）及混合（Hybird）。

Word-Aligned 充分利用了现代 CPU 的特性，即操作 Word 要比操作 Byte 效率高。所以，WAH 按照字对其的方式对 bitmap 进行分组压缩。图 2-7 展示了 WAH 的遵循的编码压缩规则：

1) Hybrid 针对 Bitmap 中既存大量连续的"0"或“1"序列，又存在"0""1"混合序列的情况，对这两种情况采用不同的压缩编码策略；
2) 将 Word 分成两类：literal words 和 fill words。其中 literal words 用于 0、1 混合序列，置最高位 bit 为 0，剩余 bits 直接存储混合序列；fill words 用于大量连续的 0 或者 1 序列，置最高位 bit 为 1，全 0 序列置次高位为 0，全 1 序列置次高位为 1，剩余 bit 存储该序列 0 或 1 的个数。高位不足则用 0 补齐。

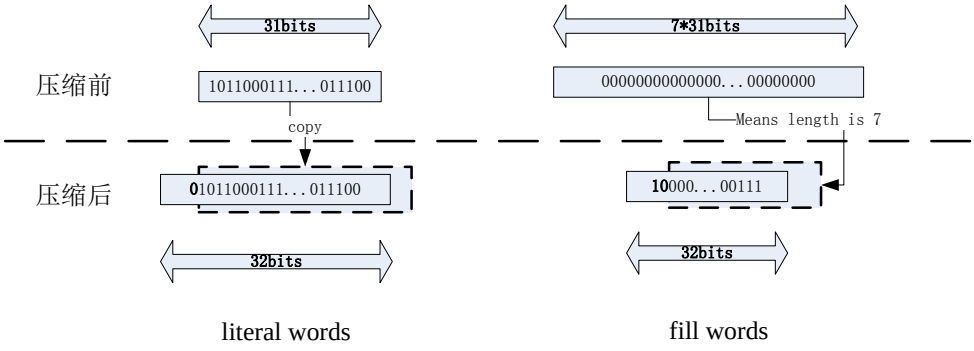


图 2-7 WAH 算法示意图

Figure 2-7 WAH algorithm diagram

WAH 压缩算法是基于游程编码（run-length encoding，RLE）原理。而 Roaring bitmap 则另辟蹊径。实验表明 Roaring bitmap 的压缩性能比 RLE 类更高。目前已知使用 Roaring bitmap 的开源项目有 Apache Spark，Apacek Lucene 和 Druid.io。Roaring Bitmap 的主要思想是将 32bit 整数分割成 2^{16} 个数据块，所有数据块共享高 16bit，低 16bit 则使用专门的容器来存储，如图 2-9。容器可以分为以下两种：

1) 当一个数据块中的整数的个数不超过 4096 时，我们认为是一个稀疏数据块，使用 16bit/整数的有序容器存储；
2) 当一个数据块中的整数的个数超过 4096 个整数时，我们认为是一个密集数据块，使用 2^{16}bit 的 bitmap 容器进行存储

为何选择 4096 作为阈值呢？如果不区分直接使用 bitmap 容器进行存储，至少需要空间为 2^{16}bit 的空间，可以表示的整数个数为 2^{16} 个。而 2^{16} 的空间如果被用来作为有序容器（short 类型 Array），最多能表示的整数个数为：$2^{16} / 2^4 = 4096$ 个元素。显然，当元素个数小于 4096 时，有序容器更省空间；而元素个数大于 4096 时，bitmap 更省空间。

下面，我们简单介绍基于 Roaring Bitmap 的相关操作。

1) 基数计算：每种容器（位图容器、数组容器）都使用一个计数器来记录它的基数。因此，最多只需要累加 2^{16} 个计数器就可以完成。
2) 数据查找：判断一个 32bit 的整数 x 是否存在，我们首先使用二分查找搜索 $x / 2^{16}$ 关联的容器。如果访问到的是数组容器，那么我们继续迭代二分查找；如果访问的是 bitmap 容器，那么我们就访问第 m 个 bit，其中 $m= x \bmod 2^{16}$，其中 0 表示不存在，1 表示存在。
3) 数据插入与删除：对于插入或者删除一个数据，我们首先需要找到对应的容器，当对象是一个数组容器时，我们使用二分查找在线性时间内插入或者删除数据；当对象是一个 bitmap 容器时，我们设置对应的 bit 值并更新计数器。当删除一个整数时，bitmap 容器的基数可能会减少到 4096，退化

为一个数组容纳。当增加一个整数时，一个整数容器的基数可能会超过4096，而变成一个 bitmap 容器。当上述两种情况发生时，Roaring bitmap 会创建新的容器，更新计数器并丢弃旧容器。

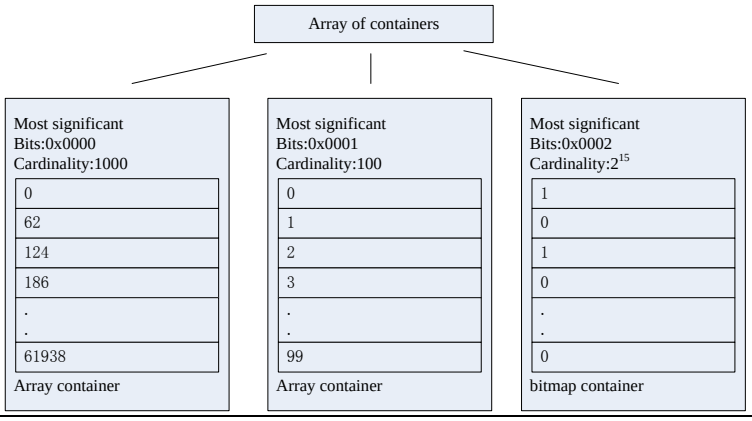


图 2-9 Roaring 算法示意图

Figure 2-9 Roaing algorithm diagram

2.3 分布式存储技术

随着互联网所承载的信息数据量的指数级增长，越来越多的互联网系统解决方案向分布式方向迁移。以 RDF 数据为代表的语义网数据，随着应用领域的不断扩展，也面临着数据量急剧增长的情况，给 RDF 数据存储系统提出了更高的要求。而分布式存储系统是由大量普通服务器通过 Internet 互联，对外作为一个整体提供存储服务，具有海量数据存储能力和并行计算能力，是解决海量 RDF 数据管理的一个有效手段。本节将对分布式存储系统的核心技术进行简要介绍。

2.3.1 CAP 定理

2000 年 7 月，加州大学伯克利分校的 Eric Brewer 教授在 ACM PODC 会议上首次提出了著名的 CAP 猜想[12]。2 年后，来自麻省理工学院的 Seth Gilbert 和 Nancy Lynch 从理论上证明了 CAP 猜想的可行性，从此 CAP 理论正式在学术上成为了分布式系统领域的公认定理，并深深的影响了分布式系统的的发展。CAP 理论阐述了这样一个事实：一个分布式系统不可能同时满足一致性（Consistency），可用性（Availability）和分区容错性（Partition tolerance）这三个基本需求，最多只能同时满足其中的两项。对这三个特性进行简要介绍：

一致性：在分布式环境中，一致性是指数据在多个副本之间是否能够保持一致的特性。在一致性的需求下，当一个系统在数据一致的状态下执行更新操作之

后，应该保证系统的数据仍然处在一致状态。

可用性：可用性是指系统提供的服务必须一直处于可用状态，对于用户的每一个操作请求总是能够在有限的时间内返回一个正常的响应结果。即保证每个请求不管成功或着失败都有响应。

分区容错性：分布式系统在遇到任何网络分区故障时，仍然能够保证对外提供满足一致性和可用性的服务，除非时整个网络环境都发生了故障。

一个分布式系统中无法同时满足上述三个需求，最多只能满足其中两项。另一方面，对于一个分布式系统而言，分区容错性可以说是一个最基本的需求，因为分布式系统中的组件必然要部署到不同的计算节点，而节点故障（服务器宕机，网络异常，磁盘故障等）或响应超时是必定会以极大概率出现的情况，因此分区容错性是分布式系统必然要面对和解决的问题。因此，分布式系统的架构设计一般需要根据需求特点在一致性和可用性之间做权衡。

2.3.2 数据分布

分布式系统区别于传统单机系统在于能够将数据分布到多个节点，并在多个节点中间实现负载均衡。数据分布的方式主要有两种，一种是哈希分布，将数据根据根据哈希值散列分布到不同的存储节点上，如 Dynamo[13]；另一种是顺序分布，将数据按照某一个键值按照整体有序的原则分布到不同的存储节点，如 Bigtable[14]。

哈希分布：如图 2-11 所示，哈希分布的实现方法一般是根据数据的某一种特征值计算哈希值，并将哈希值与集群数量建立映射关系（如 hash（*key*）mod *N*，其中 *N* 为计算节点的数据），从而将不同哈希值的数据分布到不同的服务器上。如果特征值选取不恰当容易产生数据倾斜的问题。另一个致命的问题是：当某台计算节点上线或着下线时，*N* 值发生变化，数据映射被打乱，几乎所有的数据都需要重新分布，这将会带来大量的数据迁移。一种比较成熟的方案是采用一致性哈希算法，这是分布式哈希表[15]（Distributed Hash Table，DHT）的一种实现。算法思想如下：给系统中每一个节点分配在一个随机令牌（token），这些 token 构成一个哈希环。执行数据存放操作时，先对主键进行哈希运算，然后将数据存放到顺时针方向第一个大于或着等于该哈希值的 token 所在节点。一致性哈希的优点在于节点加入或删除时只会影响到在哈希环中的相邻的节点，而对其他节点没有影响。

但值得注意的是，散列之后的数据失去了原有的整体有序性，使得对数据的操作只能支持随机读取，无法对数据进行按照顺序的方式进行扫描。并且，如果散列方式选择不恰当，可能会导致数据在节点间分布不均，损失了分布式系统的

多节点并行计算的性能。

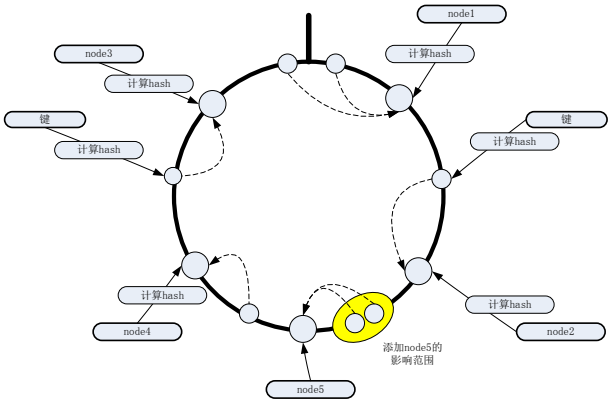


图 2-11 哈希分布

Figure 2-11 hash distribution

顺序分布：顺序分布的一般的做法是将整个数据集看作一张大表，然后将这张大表切分为若干子表。每个子表包含一定范围的有序数据。这些子表存储在不同的计算节点。如图 2-12 所示，用户表（user 表）到主键范围为 1～7000，但使用顺序分布的存储方式时，将该表按照主键划分为若干子表，子表的主键数据范围[1，1000]，[1001，2000]，…[6001，7000]。在这些子表之上，需要索引表来维护子表的位置信息以便提高查询性能。图 2-12 中，User 表的位置信息存储在 Meta 表中，Meta 表中位置信息存在 Root 表中。

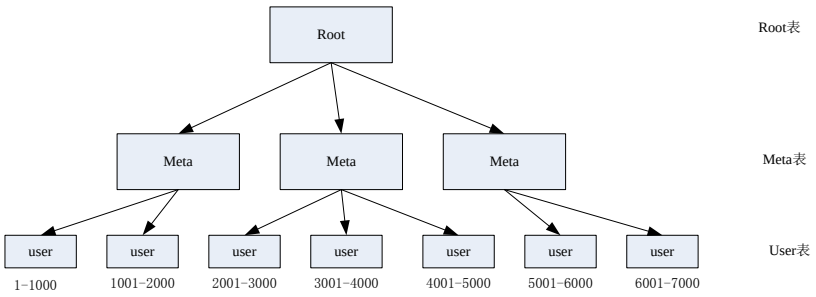


图 2-12 顺序分布

Figure 2-12 order distribution

2.3.3 范型 1:分布式文件系统

分布式文件系统的重要功能有两个：一个是存储文档，图像，视频类型数据；另一个是作为分布式表格系统的持久化层。

分布式文件系统中，最为著名的莫过于 Google File System[16]（GFS），它构建在廉价的普通服务器上，支持自动容错。GFS 内部将大文件划分为大小约 64MB 的数据块（chunk），并通过主控服务器（Master）实现元数据管理，副本管理，自

动负载均衡等操作。这里简要介绍一下 GFS 的系统架构。

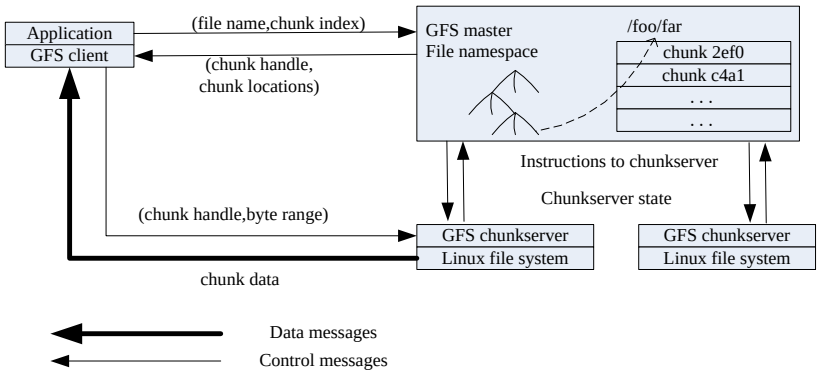


图 2-13 GFS 架构图

Figure 2-13 GFS Architecture

如图 2-13，GFS 系统的节点可分为三种角色：GFS master（主控服务器），GFS chunkserver（数据块服务器），GFS client（客户端）。

GFS 文件被划分为固定大小的 chunk，由主服务器在创建时分配一个全局唯一的 chunk 句柄（handle）。chunkserver 以普通的 linux 文件的形式将 chunk 存储在磁盘中。为了保证可靠性，chunk 在不同的节点复制多份，默认为三份。

Master 中维护系统级别的元数据，包括文件和数据块的命名空间，及其的映射信息，数据块的位置信息。同时负责对系统的全局控制，数据块租约管理，复制及垃圾回收等。

Client 是 GFS 提供给应用程序的访问接口。客户端与 GFS 交互时，首先通过 Master 获取与之相关的 chunkserver 的信息，之后直接与 chunkserver 进行通信，完成对应的读写任务。

2.3.4 范型 2:分布式表格系统

分布式表格系统对外提供表格模型，每个表格由很多行组成，通过主键唯一标识。每一行包含很多列，整个表格在系统中全局有序。Google Bigtable 是分布式表格系统的始祖，它采用双层结构，底层采用 GFS 作为持久化层。

BigTable 构建在 GFS 之上，为文件系统增加一层分布式索引层。另外，BigTable 依赖 Google 的 Chubby（分布式锁服务）进行服务器选举及全局信息维护。如图 2-14 所示，Bigtable 将大表划分为大小在 100-200MB 的子表（table），每个子表对应一个连续的数据范围。Bigtable 主要由三个部分组成：客户端程序（Client），一个主控服务器（Master）和多个子表服务器（Table Server）。

Bigtable 包含三种类型的表格：用户表（User Table），元数据表（Meta Table）

和根表（Root Table）。其中，用户表存储用户实际数据，元数据表存储用户表的元数据；如子表位置，SSTable 及操作日志文件编号，日志回放点等；根表用来存储元数据表的元数据；根表的元数据，也就是根表的位置信息，又称为 Bigtable 引导信息，存放在 Chubby 系统中。客户端，主控服务器以及子表服务器执行过程中都需要依赖 Chubby 服务，如果 Chubby 发生故障，Bigtable 系统整体不可用。

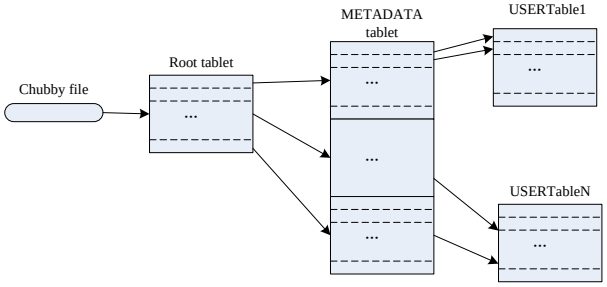


图 2-14 片位置层次结构

Figure 2-14 tablet location hierarchy

客户端查询时，首先从 Chubby 中读取根表的位置，接着从根表读取所需的元数据子表的位置，最后就可以从元数据子表中找到待查询的子表位置。

Bigtable 本质上是构建在 GFS 之上的一层分布式索引，通过它解决了 GFS 遗留的一致性问题，大大简化了用户的使用。

2.4 现有的 RDF 存储模型

当前有很多的 RDF 存储解决方案，现有的 RDF 数据存储模型主要包括基于关系型数据库的存储模型，以及非关系型数据型数据库的存储模型，本节将分别介绍这两类存储模型。

2.4.1 基于关系型数据库的存储模型

关系型数据库是建立在关系模型上的数据库，借助于集合代数等数学概念和方法来处理数据库中的数据，具有非常强的关系完整性约束。各种实体之间的各种联系均用关系模型来表示。当进行数据查询时，将 RDF 查询转化为关系型数据库查询（如 SQL 等）。代表系统有 Jena，RDF3X，3store[25]，包括基于 MySQL 作为后端的 Cayley 等。

关系型数据库由于其成熟的技术和稳定高效的性能，一直以来是持久层介质的优选方案，也是最先被应用到 RDF 存储解决方案中来的。但关系型数据库模型与 RDF 数据模型往往是不一致的，换句话说，关系型数据的关系模型是二维表格

模型，而 RDF 数据模型是图数据模型。在存储和查询时需要做一层转换，因此带来比较大的开销。也是当前研究的优化方向之一。因此，基于关系型数据库进行存储，需要将图模型的数据进行分割，转化二维表模式。现有的有垂直表，属性表和水平表三种模式。以下分别介绍这些存储模式:

(1) 垂直表存储

Subject	Predicate	Object
Id_1	publicationType	Survey Paper
Id_1	hasTitle	“Querying RDF data”
Id_1	authorBy	Id_2
Id_2	hasName	John
Id_2	hasEmail	John@cse.unsw.edu.au

图 2-15 三元组垂直表存储模式

Figure 2-15 triple vertical table storage mode

垂直表是将三元组直接存储在一张表中，该表由主语，谓语和宾语三列构成，如图 2-15 所示。如果表的规模较大，可以对其进行垂直分割，切分为同样模式的表。这是一种非常直观的存储模式。RDF-3X 采用的垂直表的存储模型。

显然，三元组的垂直存储方案的结构非常简单明了，数据存储时不需要做过多的转换工作，可直接插入存储，易于在数据库上实现，而且规模较大时，垂直扩展较为简单。但这样带来的一个问题是查询处理效率较低。所有的数据都集中存储于一张或几张表，数据量很大时导致表规模过大，虽然可以通过索引进行优化，但查询的响应性能依然较低。

更为严重的是，当主语相同而属性不同的三元组存储在一张表的不同行时，把一个 RDF 的 SPARQL 查询转换成 SQL 查询时会触发大量的自连接（join）操作，例如下面的查询:

```
select ?z where{
 ?x bookTitile "Graph Theory" .
 ?x bookType "Textbook" .
 ?x bookAuthor ?y .
 ?y bookWeb ?z .
}
```

转化为 SQL 查询为：

```
select T4.object
   from Triples as T1，Triples as T2，Triples as T3，Triples as T4
   where T1.Predicate = "bookTitile" and T1.Object = "Graph Theory" and
```

T2.Predicate = "bookType" and T2.Object = "Textbook" and
T3.Predicate = "bookAuthor" and
T4.Predicate = "bookWeb" and
T1.Subject = T2.Subject and
T1.Subject = T3.Subject and
T3.Object = T4.Subject

通过上面转化后的 SQL 查询语句可以看到，该查询会引起大量的自连接操作。如果数据表的规模超过一定程度，自连接的增加会显著影响查询效率。因此在这种情况，必须通过增加索引的方式提高查询效率。

针对这种情况，Hexastore[24]利用“空间换时间”的方案对垂直表进行性能优化。在三元组表存储的基础上，根据 s，p，o 的顺序建立{spo，sop，pos，pso，osp，ops}六个索引，以其中的 spo 索引为例，该索引是以主语 s 为第 1 顺序列，以谓语 p 为第 2 顺序列，以宾语 o 为第 3 顺序列而建立的三维索引。通过增加索引，有效的提高了查询效率。但同样引入了存储开销膨胀的问题，而且在索引的创建过程和数据的修改操作会更为复杂。

RDF-3X 在 Hexastore 六个索引的基础上由进行了优化，使用了 3 个一维{s，p，o}索引及 6 个二维索引{sp，ps，so，os，po，op}。为了降低增加索引带来的存储压力，RDF-3X 通过对索引进行压缩来降低存储开销。

(2) 属性表存储

属性表存储将相同主语的 RDF 三元组数据存储为一行，该行的其他列名为该主语代表的实体的属性，列的值即为对应的属性值，如图 2-16 所示。Jena 采用的就是属性表存储模式。

Publication

ID	publicationType	hasTitle	authoredBy	editedBy
Id_1	Survey Parper	“Querying RDF Data”	Id2	Id3

Person

ID	hasName	hasEmail	webPage	roomNo
Id_1	John	John@cse.unsw.edu.au		518

图 2-16 三元组属性表存储模式

Figure 2-16 triple attribute table storage mode

如果只采用一张属性表来存储所有的 RDF 数据，很容易看出来，表的列维度会非常的大。因为不同的实体拥有的属性是不一样的，导致表中出现很多空值，整个表具有非常大的稀疏性，引起严重的空间浪费。

为了解决这个问题，需要将一张属性表按照某种方式划分为多个属性表，使得存储空间能够紧凑起来。一般来讲，可以直接通过先验知识或者聚集算法将多个有关联关系的属性组织在一张表中。

属性表存储方式解决了垂直表存储引起的查询时自连接操作过多的问题。由于同一个主语的多个属性值存储在表的同一行，在执行转换后的 SQL 查询后显著减少了自连接操作，以如下 SPARQL 查询为例：

```
select ?z where{
  ?x bookTitile "Graph Theory" .
  ?x bookType "Textbook" .
  ?x bookAuthor ?y .
  ?y bookWeb ?z .
 }
```

转化为 SQL 查询为：

```
select Person.webPage
    from Person，Publication
    where Publication.bookType = "textbook"
        and Publication.hasTitile = "Graph Theory"
        and Publication.bookAuthor = Person.ID
```

但同时属性表存储方式也具有明显的劣势：

1) 属性列扩展能力差。如果对属性表增加一个新的属性值，则需要对表进行列修改。对关系型数据库而言，列的修改将引起表模式（schema）的修改，涉及到该表的其他数据的也会随之改变，这一操作的代价非常昂贵；
2) 对多值属性的解决能力差。如果属性表中某个主语的属性对应多个值，这个属性称为多值属性，属性表对多值属性的存放不容易解决，往往使用一些不太优雅的手段；
3) 无法彻底解决空值问题。通过属性划分之后，属性表一定程度上解决了存储模式导致的空值上的空间浪费，但是数据本身的稀疏性无法彻底解决，因此属性表仍然存在一定程度的空值；

(3) 水平表存储

水平表的存储方式如图 2-17 所示，它将谓语相同的三元组存储到同一张表中，由于表的谓语相同，因此不需要对谓语进行存储，只存储主语和宾语两列。这种方式即为根据谓语对三元组进行垂直划分。通过这种方式，每个属性表对应一张二元表。RDF 数据中谓语的个数即为数据库中表的个数。

publicationType

ID	value
Id_1	Survey Parper

hasTitle

ID	value
Id_1	“Querying RDF Data”

authoredBy

ID	value
Id_1	Id2

图 2-17 三元组水平表存储模式

Figure 2-17 triple horizontal table storage model

水平表存储方式较好的解决了之前模式存在的两个问题：

1) 解决了多值属性的存储问题。当面对多值问题时，水平表将其存储在不同的行中，高效的支持了多值属性存储；
2) 解决空值存储引起的空间浪费问题。水平表存储属性值存在的三元组，对空值的数据不会进行存储，避免了空值存储引起的空间浪费。

同时，水平表的存储也存在一定的劣势：

1) 表的数量难以控制。水平表根据谓语的进行分割，而真实数据集中包含大量的不同谓语，这样导致水平分割后的表数目过多，带来维护上的难度；同时在查询时，可能会引起大量的表之间的连接操作，影响查询性能；
2) 当查询涉及到属性未知时，查询效率急剧下降。因为当谓语未知时，需要对所有数据进行查询，即进行全库扫描（scan），严重影响查询性能。

2.4.2 基于非关系型数据库的存储模型

关系型数据库的存储模型基于关系模型的强约束，虽然能够通过水平或者垂直分割方式解决模型转化上的问题，但是随着数据量规模的膨胀触发一系列影响性能的问题。而且尽管一些关系型数据库能够通过一定的方式支持分布式部署，但其扩展能力有限。而近年来比较流行的非关系型数据存储模式，由于不存在强的关系约束，具有天然的分布式部署支持，如 Bigtable，HBase[21]等，并且拥有与关系型数据相当的性能，能够存储海量的 RDF 数据。学术界和工业界也相继出现各种系统解决方案，基于非关系型数据库的分布式系统进行海量 RDF 数据的存储和管理，并利用分布式计算引擎的并行计算能力，对查询等一系列操作进行并行处理。下面将简单介绍一些基于非关系型数据库的 RDF 分布式存储解决方案。

文献[17-18]采用了直接将 RDF 数据以文件形式存储在 Hadoop[27]分布式文件

系统中，并采用 MapReduce[22]计算引擎对 SPARQL 语句进行并行处理。在数据存储层，文献[17]依据属性及宾语所属的类型将文件分块存储；而文献[18]则依据主语对文件进行划分；在逻辑查询层，两个方案根据不同的算法生成 MapReduce 任务，对查询进行迭代计算。

文献[19-20]采用的是基于 HBase 的 RDF 存储方案。文献[19]在存储层采用了类似关系型数据存储时采用的垂直表存储的方式，并建立多个索引，将数据存储在 SPO，POS，OSP 三张表中，以 SPO 表结构为例，该表以（主语，谓语）为 row-key（行键），宾语为 value（键值），使用 HBase 的用多个列来处理多值问题；文献[20]分别以 S，P，O 为 row-key，PO，SO，SP 为 value 组成三张表 TS，TP，TO 来对数据进行存储。在查询逻辑层时，利用 HBase 提供的应用程序接口实现了支持 SPARQL 查询。

文献[21]是基于 Accumulo 数据库存储 RDF 数据，在存储层和查询逻辑层采用的方式与文献[17-19]相似。

从上面的这些系统实现方案可以看出，基于非关系型数据库的分布式 RDF 存储方案在存储层通常是直接基于文件形式，将数据组织成多索引表的结构进行存储。在查询逻辑层，则是依赖于系统提供的应用程序接口，生成对应的查询任务，执行查询计算。

2.5 本章小结

本章主要介绍了语义网的相关概念：资源描述框架（RDF），及对应的查询语言（SPARQL）。并且介绍的主流的单机及分布式存储技术，着重介绍了位图索引技术及主流的压缩算法。同时给出了当前 RDF 集中式存储和分布式存储的一些设计方案。

3 RDF 存储及索引结构设计

本章将介绍围绕 RDF 存储模型及索引结构设计展开。首先概述系统设计的目标和整体方案，之后给出底层存储和索引结构的设计细节，以及对应的数据载入和获取逻辑，最后对本章进行小结。

3.1 设计概述

本节主要围绕设计目标进行论述。设计目标给出了系统要达到的基本要求，通过对基本要求分析，定位系统设计中的关键技术难点，结合上一章的相关技术，选择和构建合适的技术方案。同时对一些关键技术问题，本节也进行了讨论。

3.1.1 设计目标

该存储引擎的目标是实现一个能够支持 GB 数据量的，可扩展为分布式的单机系统原型。在设计上有如下要求：

(1) 该系统在实现单机部署的基础上，后期可以通过一致性保证的中间件，实现分布式的水平扩展；

(2) 支持较高的查询率，读取性能要求要高于写入新能要求，在保证高响应的基础上，尽可能节省内存资源，压缩静态使用空间；

(3) 尽可能降低硬盘访问次数；尽可能面向 SSD 的数据访问方式，为硬件升级提供平滑的演进。

3.1.2 整体设计方案

根据设计目标的要求，有以下几个问题需要思考：

(1) 在不借助第三方中间件（如 Hadoop,HBase 等）的基础上，如何做到能底层的存储模式支持分布式的水平扩展？

(2) 支持分布式必然牵扯到文件的切分，切分是按照垂直方式，水平方式还是属性方式？切分完之后的文件块按照何种方式进行存储，是顺序存储还是哈希存储？

(3) 支持较高的读性能，必然需要对数据进行索引，索引引擎是采用哈希存储引擎，B+树存储引擎，LSM 树存储引擎，还是位图存储引擎？哪种更适合该系统设计要求？

首先对于第一个问题，如何支持分布式的水平扩展性。在这一要求上，可参考 Hadoop 文件系统以及在其之上的 Hbase 的系统设计方案。HBase 通过在 Hadoop 之上增加了提供一致性保证的中间件实现了分布式表格系统的扩展。因此在底层的文件存储结构之中，可采用类似 Hadoop 的文件分块方案，将数据文件切分为多个块，并通过这些文件块的元数据文件对文件块进行组织和管理。至于文件块的是进行顺序存储还是哈希存储，则跟文件块的分块原则相关。

对于第二个问题，关于确定文件块的切分原则。通过对基于关系型数据库的 RDF 数据管理模式的总结可以看出，文件块的切分可以按照水平，垂直以及属性来进行切分。属性表的扩展能力太差，不适合后期分布式的扩展；水平表当面对谓语未知时，需要全表扫描，这一点对于面向分布式的系统来讲是一个会严重影响系能的问题；而垂直切割的方式在操作性上最为简单，维护性最好。采用垂直切割的方式，文件内容按照的原始相对位置，直接进行顺序存储。

对于第三个问题，在资源一定的情况下，设计一个满足要求的可依赖系统面临的都是一个时间与空间的权衡（trade-off）。要求系统具有较高的查询性能，则必然需要在数据内容的基础上进行索引构建，以消耗一定的存储空间为代价换取响应时间的性能提升。值得注意的是，RDF 三元组的查询是面向三个位置（主语位置，谓语位置，宾语位置）的查询，因此，三个位置都需要进行建索引，这对于存储空间的消耗带来比较大的负担。

通过之前的数据库引擎可以看出，LSM 树是为优化写性能而设计的，与本设计面向读的系统设计目标矛盾。哈希存储引擎查询的时间复杂度最低，但哈希引擎使用的哈希表结构对内存的消耗较大，而且不易调优。单纯使用哈希表对所有数据进行索引，在面对数据量较大时，系统根本无法承受。B+树存储引擎实现较为复杂，而且不太容易进行分布式的数据扩展。位图索引高效的查询和计算特点，以及可支持压缩的特性非常符合设计要求，并且位图索引能够很方便的进行分布式的扩展。在位图的压缩算法方面，本设计采用了当前性能评价最好的 Roaring 压缩算法，能够最大程度降低运行时的内存消耗。

为了降低查询时的硬盘 IO 操作，需要对每条三元组数据通过其物理存储位置的偏移量进行标示。因此，每个存储块中不仅包含原始数据，还要包含每条三元组的偏移量信息，当进行读取对应行的数据时，通过二分查找的方式进行位置搜索。

总而言之，在存储层采用了类似 HDFS 分块存储的方式，对数据进行垂直切分，通过元数据结构对分块数据进行组织，以提供系统的扩展性；同时使用位图的方式对主语，谓语和宾语进行索引，并采用 Roaring 压缩算法对位图进行压缩，以节省空间。

下面将对底层存储结构的设计进行较为详细的描述。

3.2 底层结构设计

底层存储参考了 HBase 底层顺序存储做法。将全部的三元组行数据文件切分为具有固定大小的文件块，即对所有的三元组文件采用无语义垂直分割的方式直接存储，然后通过多级索引进行组织。

如图 3-1 所示，所谓的三元组行数据文件，即三元组的数据直接以明文按行存储，主语 s，谓语 p，宾语 o 以制表符（"\t"）间隔开，以换行符（“\n"）区分不同三元组。

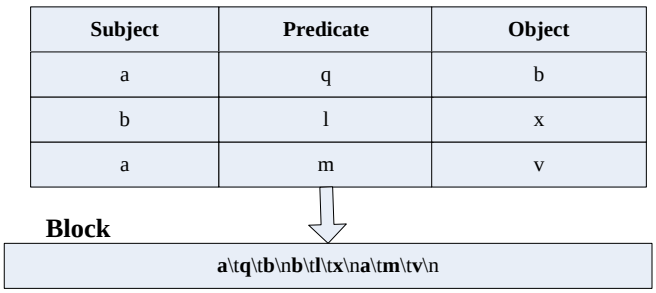


图 3-1 RDF 底层存储格式

Figure 3-1 RDF low level storage format

如图 3-2 所示，将整个 RDF 数据文件顺序划分为连续范围的行数据块，每个数据块称为一个 Block，整个 RDF 数据共有 4531 行，因此行坐标范围为 0～4530。在底层存储系统中划分为多个 Block，分别对应数据行范围[0，88]，[89，101]，…[4353，4521]，[4522，4530]内的数据。每一个 Block 都有一个对应的 Block_Index，该 Block_Index 中存储了该 Block 中每一行的数据(即每一条 RDF 数据)在该 Block 中的偏移量，因此 Block_Index 为该数据的一级索引层。众多的 Block_Index 通过 Root_Index 进行组织和管理，Root_Index 中记录了每个 Block_Index 的信息（该 Block_Index 包含的行偏移范围），因此 Root_Index 为该数据层的二级索引。

需要注意的是，底层存储层仅提供通过全局行坐标来读取指定 RDF 行数据的功能，并不包含任何语义特征的读取功能（如按照某个字段）。因此当读取制定行的 RDF 数据时，先通过二级索引层 Root_Index 查找到对应的一级索引层 Block_Index，然后通过一级索引层 Block_Index 查找到指定行的位置信息，最后通

过位置信息发起硬盘读取。

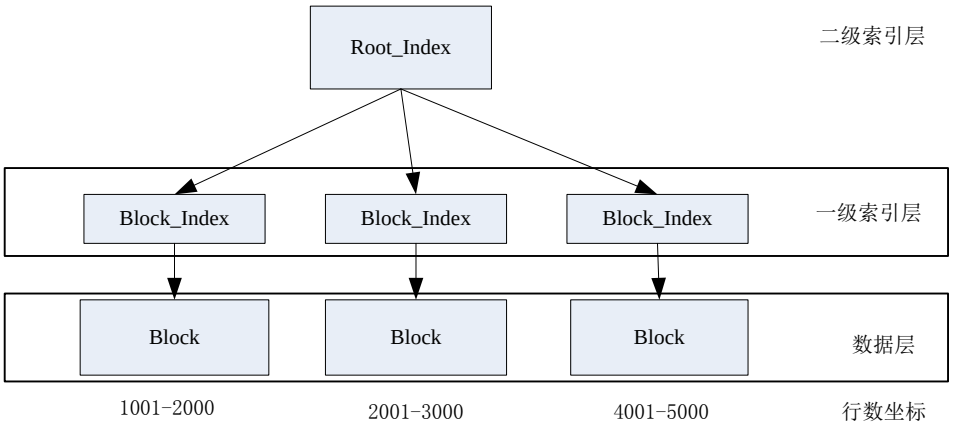


图 3-2 RDF 底层存储模型

Figure 3-2 RDF underlying storage model

这样的设计目的出于支持分布式水平扩展的需要。如果 RDF 数据文件过大，单机无法承载时，可以将 Block 按照一定的策略分配到不同的存储节点上，为了支持更大的集群规模，将索引分为两级：一级索引层 Block_Index 和二级索引层 Root_Index，由 Root_Index 表维护 Block_Index 表的位置信息，而 Block_Index 表维护 Block 的位置信息。这样可实现分布式支持的平滑扩展。

下面将简要介绍各个组件的实现方案。

3.2.1 持久层分块数据结构

如图 3-23 所示，持久层分块数据结构 Block，包括数据头部信息，数据本体等。

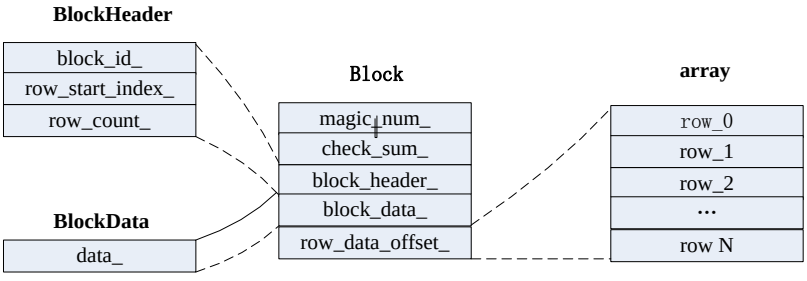


图 3-3 Block 结构

Figure 3-3 block structure

Block 数据结构定义如下：

```
struct alignas(8) Block{
 private:
```

```
    char magic_num_[8];
    char check_sum_[8];
    BlockHeader block_header_;
    BlockData block_data_;
    uint64_t row_data_offset_[1]; //flexiable array，whose length in block_header_
};
```

BlockHeader 数据结构定义如下：

```
struct alignas(8) BlockHeader{
 private:
    uint64_t block_id_;
    uint64_t row_start_index_;
    uint64_t row_count_;
};
```

BlockData 数据结构定义：

```
struct BlockData{
 private:
    char data_[BLOCK_DATA_SIZE_LIMIT];
};
```

其中，magic_num_用于该 Block 数据结构持久化（dump）及载入（load）时的类型的校验字段。check_sum_字段用于校验文件的完整性。block_header_是一个 BlockHeader 结构类型数据，包含三个字段：block_id_为该 Block 的全集唯一 ID，row_start_index_为该 Block 中第一行数据在全局数据的位置（如 47，则说明该 block 的第一行在整个大文件中为第 47 行）。row_count_表示该 Block 中含有多少行数据。继续回到 Block 定义中，block_data_为 BlockData 类型数据，用与存放该 Block 中的三元组数据。值得注意时，row_data_offset_字段为一个柔性数组（flexiable array）。因为在切分文件是按照固定大小（即最大为 BLOCK_DATA_SIZE_LIMIT）进行的，但每一个 Block 数据中的行数是不同，因此，在 BlockHeader 的 row_count_记录该 Block 行数，同时也是该柔性数组的实际长度。

3.2.2 持久层根表数据结构

持久层根表数据结构体 RootTableSpec 定义如下：

```
struct RootTableSpec{
```

```
  private:
    char magic_num_[8];
    uint64_t list_size_;
    uint64_t data_[1]; //flexible array
};
```

RootTableSpec 对 RootTable 持久化及载入时需要的数据结构。该结构包含了构建 root table 需要的必要信息。同 Block 结构体一样，magic_num_为文件类型的校验字段。list_size_表示该文件拥有的 block 的数量。data_字段中存放每个 block 的第一行数据所在全局的位置（即每个 block 的 row_start_index_字段），其顺序按照 block 的 id 升序存放。显然，data_的长度即 list_size_。

3.2.3 运行时块索引结构

运行时块索引结构 BlockIndex 定义如下：

```
class BlockIndex{
  using IndexType = Block::row_data_offset_type;
  public:
    BlockIndex(const std::string& blockfile_name);
    ... ...
    IndexType at(size_t row_num) { return index_[row_num]; }
    ... ...
  private:
    uint64_t block_id_;
    uint64_t row_start_index_;
    uint64_t row_count_;
    std::vector<IndexType> index_;
};
```

BlockIndex 是对应的持久化结构体 Block 在运行时构建的索引结构，其中的字段也相应的对应 Block 中的字段含义。BlockIndex 有两个重要的接口：

(1) 构造函数 BlockIndex（const std::string& blockfile_name）是通过 Block 持久化后的文件实现初始化，即读入解析 Block 文件;

(2) IndexType at（size_t row_num）：row_num 为全局行坐标，返回值为该行在该 Block 中的索引值（偏移量）。

3.2.4 运行时根表数据结构

运行时根表数据结构 RooTable 定义如下：

```
class RootTable{
 public:
   ... ...
   uint64_t get_block_index_by_global_offset（uint64_t global_offset）;
   int get_seek_pos_by_row_index（uint64_t row_index，uint32_t& ret_block_id，
                                  uint64_t& ret_block_offset）;
   ... ...
 private:
   std::vector<uint64_t> block_offset_list_;
   std::vector<std::shared_ptr<BlockIndex>> block_index_list_;
   std::string table_name_;
};
```

相应的，RootTable 是 RootTableSpec 运行时构建的结构，其中的字段与 RootTableSpec 对应。RootTable 有两个重要的接口：

(1) uint64t get_block_index_by_global_offset（uint64t global_offset）：通过全局行偏移量（global_offset）确定该偏移量归属的 block_index 的 id；

(2) int get_seek_pos_by_row_index（uint64t row_index，uint32t& ret_block_id，uint64_t& ret_block_offset）;通过全局的行偏移量 row_index 确定其所在的 block_id 和其在该 block 中的偏移量。

3.2.5 运行时块数据句柄结构

运行时硬盘块数据读取句柄 BlockDataSeeker 结构定义如下:

```
class BlockDataSeeker{
 public:
   BlockDataSeeker（const std::string& block_file_name）;
   ... ...
   int get_triple_by_index（uint64_t index_offset，
                            std::shared_ptr<core::TripleSpec> p_triple_spec）;
 private:
   int fd_ ;
```

```
    ... ...
};
```

当进行硬盘读取时，需要借助 BlockDataSeeker 根据偏移量来读取三元组数据。fd_保存的是该 block 文件的句柄。构造函数是通过传入 block 文件名进行初始化，其中有一种重要的读取接口：

(1) get_triple_by_index（uint64_t index_offset，std::shared_ptr<core::TripleSpec> p_triple_spec）通过传入该三元组数据的偏移量（index_offset）实现读取，返回的三元组数据由 p_triple_spec 存储。

3.2.6 运行时三元组数据结构

运行时三元组数据结构 TripleSpec 定义如下：

```
class TripleSpec{
 public:
  ... ...
  const std::string& at（TripleELemPos pos） const;
 private:
  std::string sub_;
  std::string pre_;
  std::string obj_;
 };
```

运行时三元组结构设计比较简单，以字符串形式存储主语（sub_），谓语（pre_），宾语（obj_）。并提供了通过三元组中的位置获取相应字符串的接口。

3.3 索引结构设计

底层存储结构提供了按照行坐标读取行 RDF 数据的接口，我们需要基于底层存储结构的基础上，设计和实现面向关键词（term）的查询机制，来支持语义搜索。

举个例子，比如，当我们有如下 SPARQL 描述搜索请求时：

```
select ?name where{
      id:00001 people.name ?name .
}
```

我们需要通过关键词索引来获取主语为“id:00001”且谓语为“people.name”的 RDF 三元组的行坐标（这里可能有多个符合要求的三元组，因此会获取多个行

坐标），然后通过底层存储结构提供的数据读取接口，发起硬盘 IO，获取对应的三元组，即可获得对应的 name 结果。

因此，该节的“索引”与 3.2 节底层存储结构的“索引”并不是同一个概念。3.2 节底层存储结构中的“索引”是将 RDF 三元组行坐标映射到 RDF 三元组的数据内容，是存储层索引；而该节的“索引”是将关键词映射到 RDF 三元组行坐标，是语义层索引（类似于全文检索中的倒排索引），如图 3-4 所示。

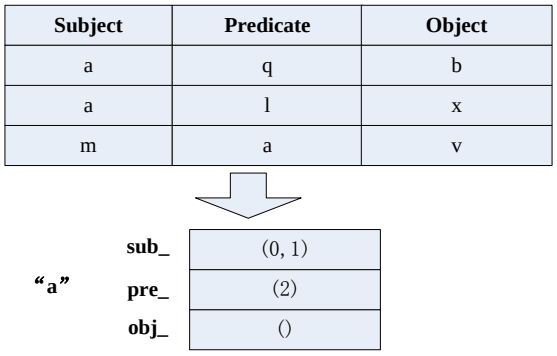


图 3-4 RDF 索引结构

Figure 3-4 RDF index structure

在图 3-4 中，RDF 三元组数据共有三行，（a，q，b），（a，l，x），（m，a，v）。其中，我们以 term “a”为例，它的索引内容类似为箭头下方的数据结构。该索引的含义：以“a”为主语的三元组为行坐标为 0，1 的数据，以“a”为谓语的三元组为行坐标为 2 的数据，以“a”位宾语的三元组不存在。在系统实现中，索引的组织形式和数据结构没有像图 3-4 中那样简单，在系统实现中以将位图的形式存储每个 term 在三个位置上的索引，并通过 Roaring 压缩算法对位图进行数据压缩，以降低运行时内存占用空间。

3.3.1 索引数据类型

方案将运行时的可索引数据（即 term）的类型分为两类：哈希字符串（HashValue，64-bit 的 16 进制表示，来自于 128bit 的 md5 值的中间部分）和普通短字符串（ShortString）。

```
namespace IRIType{
  class Value{};
  class HashValue : public Value{
    using value_type = uint64_t;
    ... ...
    private:
```

```
    value_type value_;
};
class ShortString : public Value{
    using value_type = std::string;
    ... ...
    private:
      value_type value_;
  };
}
```

从上面定义可以看到，为了节省运行时存储空间，HashValue 使用 uint64_t（8-bytes）类型来存储数据，相比较直接用字符串（16-bytes）节省了 50%的内存占用。在 ShortString 的定义中，采用了定义索引单词的经验方法[32]：单词是数字和字母的最大序列，但总长不超过 256 个字符（character）。因为在系统查询中对术语是进行完全匹配的，因此过于长的字段暂时认为是没必要进行索引的。

3.3.2 整体索引结构

在运行时的数据索引结构也是通过二级方式组织，与图 3-2 所示的底层存储模型相对应。运行时索引结构如图 3-5 所示。

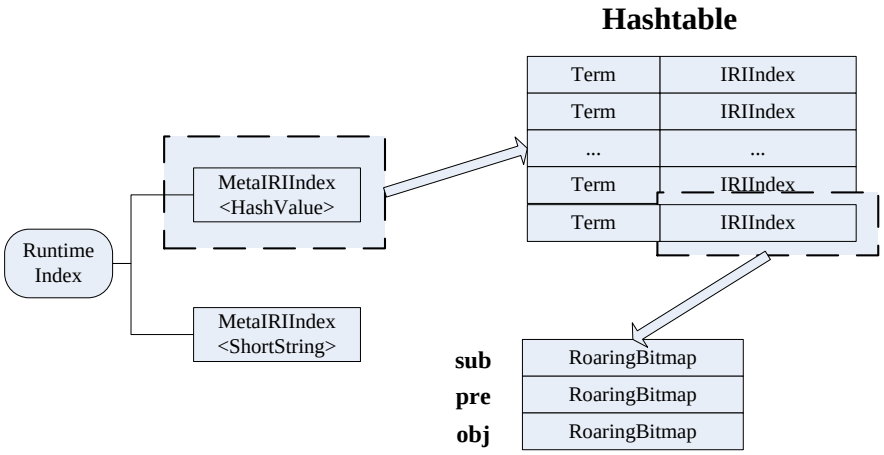


图 3-5 运行时索引结构

Figure 3-5 runtime index structure

索引分为针对每个术语的索引结构（IRIIndex）和元索引结构（MetaIRIIndex）。

(1) 针对每个术语的索引结构

IRIIndex 结构定义如下：

```
template<typename IRIType>
```

```
class IRIIndex{
 using BitMap_T = Roaring64Map;
 public:
  ... ...
  const BitMap_T& sub_index（） const {return sub_index_;}
  const BitMap_T& pre_index（） const {return pre_index_;}
  const BitMap_T& obj_index（） const {return obj_index_;}
  .... ...
  const BitMap_T& at（TripleELemPos pos） const ;
 private:
  IRIType value_;
  BitMap_T sub_index_;
  BitMap_T pre_index_;
  BitMap_T obj_index_;
};
```

每一个术语（term，即被建索引的单词）的索引结构中，包含三个 bitmap，即该术语在主语中的出现位置的 bitmap（sub_index_），在谓语中出现位置的 bitmap（pre_index_），在宾语中出现位置的 bitmap（obj_index_）。该索引结构如图 3-4 所示。

Subject	Predicate	Object
a	q	b
b	l	x
a	m	v
c	w	d
z	n	a
b	s	m
s	a	u
k	b	a

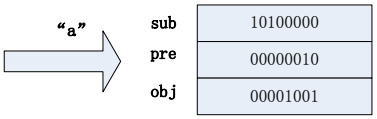


图 3-4 term 索引结构

Figure 3-4 term index structure

一般来讲位图索引是比较稀疏的。因此会进行压缩，以降低运行内存资源的占用。这里我们采用了 Roaring 压缩方法。同时，我们提供了访问接口，根据位置获取对应的位图。

值得注意的是，term 的位图索引是在第一次读入 RDF 数据时，同数据分块存储一起构建的，因此也是会持久化到硬盘上。因此每次启动数据库服务进程时，直接从硬盘中载入，不需要重新组织。

(2) 元索引结构

MetaIRIIndex 结构定义如下：

```
template <typename T>
class MetaIRIIndex{
 using MAP_KEY_T = typename IRITypeTrait<T>::value_type;
 using MAP_VALUE_T = typename std::shared_ptr<IRIIndex<T>>;
 using MAP_TYPE =
        typename std::unordered_map<MAP_KEY_T，MAP_VALUE_T>;
 public:
        ... ....
 private:
      MAP_TYPE meta_iri_map_;
 };
```

元索引结构利用散列表对单个术语索引进行组织。这样能够以 O（1）代价获取对应的术语索引。读取术语对应索引时，首先根据元数据索引 MetaIRIIndex 获取 IRIIndex 的指针，然后通过 IRIIndex 可获得其在不同位置上的索引。

3.4 数据载入和获取逻辑

本节简要介绍如何如何将 RDF 数据载入进而组织成底层数据结构，以及如何从底层数据结构中将 RDF 数据读出。

3.4.1 数据载入

将 RDF 数据按行读入，并同时更新术语索引。当满足一个 block 大小时，将该 block 持久化到硬盘。全部文件被切分和构建索引完成后，将索引数据持久化到硬盘。载入流程如表 3-1 所示。

```
TRIPLE_LOAD():
1:   BLOCK_SIZE := 128MB
2:   iri_index := empty_map
3:   current_block_size := 0
4:   current_block := empty_block
5:   for each row_rdf_triple in raw_rdf_file:
6:      current_row_size := the length of row_rdf_triple
7:      if current_row_size + current_row_size > BLOCK_SIZE:
8:         dump current_block and block_index
9:         current_block := empty_block
10:        current_block_size := 0
11:        update root_index
12:     row_index := current row index
13:     sub, pred, obj = parser row_rdf_triple
14:     update iri_index of sub, pred, obj
15:     append row_rdf_triple to current_block
16:     current_block_size += current_row_size
17:     update block_index
18:  dump iri_index into disk
```

表 3-1 三元组数据载入过程

Table 3-1 triple data loading process

该算法非常直观。简而言之就是数据分块的同时，构建顺序存储行索引和关键词索引，并进行持久化。

3.4.2 数据获取

在这一层面，我们提供了根据全局行坐标来获取对应三元组的功能和根据术语来获取对应的 bitmap 索引功能。这两个功能保证了我们查询数据时必要功能。

我们采用下面这个列子，展示如何通过底层存储结构和关键词索引结构获取制定关键词的 RDF 三元组数据。

Subject	Predicate	Object
a	q	b
b	l	x
a	m	v
c	w	d
z	n	a
b	s	m
s	a	u
k	b	a

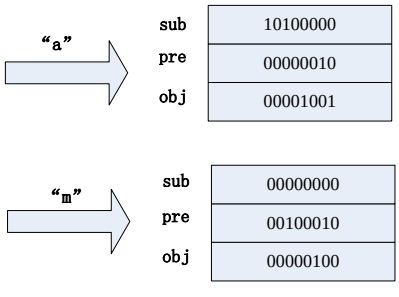


图 3-5 RDF 数据及其 term 索引

Figure 3-5 RDF data and term index

有如图 3-5 左侧所示的 RDF 三元组数据，并在右侧展示了其中两个关键词的索引：“a”和“m”。底层的存储结构在这里不再展示。如果查询如下 SPARQL 请求时：

```
select ?obj where {
        "a" "m" ?obj .
}
```

获取过程如下：

(1) 根据关键词“a”，获取该关键词对应对 IRIIndex，“a”处在主语位置，因此获取“sub”字段对应的 bitmap，令其为 bm_1，则 bm_1=10100000；

(2) 根据关键词“m”，获取该关键词对应对 IRIIndex，“a”处在谓语位置，因此获取“pre”字段对应的 bitmap，令其为 bm_2，则 bm_2=00100000；

(3) 将 bm_1，bm_2 进行取交集，即进行逻辑与（&）计算，结果为 bm_3，则 bm_3=bm_1&bm_2=（10100000）&（00100000）=（00100000）

(4) bm_3 即为目标 RDF 三元组数据的位图，取置 1 位，即第三行。通过底层行存储接口，读取第三行数据，获取对应的该行三元组的宾语即为?obj 的结果：“v”。

3.5 本章小结

本章主要介绍了 RDF 底层存储设计和实现，即基于顺序存储的方式，并提供二级索引，以支持通过行坐标来读取三元组数据；同时介绍了支持语义查询的关键词索引的设计和实现，即基于 Roaring 压缩的位图索引，为每个关键词提供 3 个位置的索引（主语，谓语，宾语）。并阐述了基于该设计的数据载入和获取逻辑。

4 RDF 查询逻辑设计

这一章主要介绍面向 SPARQL 的 RDF 数据查询逻辑的设计和实现。

4.1 查询逻辑概述

在整个查询层，面向外部的查询接口接收 SPARQL 查询语句。如图 4-1 所示，总体执行流程如下。

(1) **查询触发:**客户端调用查询接口，传入 SPARQL 查询语句；

(2) **SPARQL 解析**：将 SPARQL 语法解析模块转换成对应的查询单元集合；

(3) **查询图构建与执行**：根据一定的策略对依次从查询单元集合挑选出查询单元，执行计算。这些计算对应存储层的数据查询动作；

(4) **结果图构建:** 执行完成之后，将获得的图结构数据转换成二维表结构数据，回复给调用查询接口的客户端。

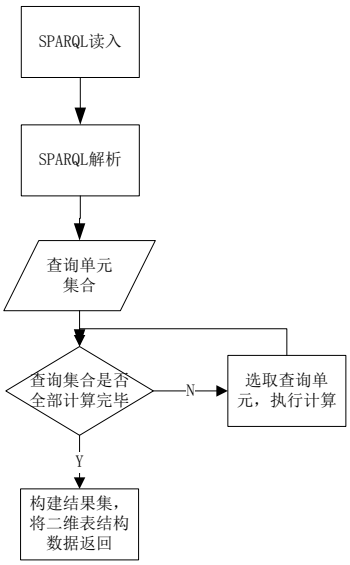


图 4-1 查询逻辑流程图

Figure 4-1 Query logic flow chart

4.2 SPARQL 语法解析

根据 SPARQL 语法，实现了一个能够解析为对应内部数据获取接口的语法解析模块。该模块主要实现以下关键点：

(1) 解析出需要获取哪些变量（variables）；

(2) 解析出存在哪些三元组约束，并如何将这些约束合理的映射为查询接口。

SPARQL-parser 这部分在工程实现上比较繁琐，详细介绍将会占据大量篇幅，而且并不是该论文的重点，我们将会简要概述，仅介绍背后的设计实现思路。

4.2.1 三元组语义类型

在非变量（variable）类型上，沿用了底层存储的 HashValue 和 ShortString 类型；在变量类型上，定义了字符串类型的 Variable 类型变量。即

```
namespace query{
 using HashValue = core::IRIType::HashValue;
 using ShortString = core::IRIType::ShortString;
 using Variable = std::string;
 ... ....
}
```

比如，当面对这样一条 sparql 语句：

```
select ?name where{
 <rdfs://45ac92yhbf1109ac> <rdfs://type.object.name> ?name .
}
```

需要将“<rdfs://45ac92yhbf1109ac>”转化为 HashValue 类型变量，“<rdfs://type.object.name>”转化为 ShortString 类型变量，而“?name”则转化为 Variable 类型变量。

我们将每一条三元组查询语句作为一个查询单元，即 TripleQuery 结构体：

```
class TripleQuery{
 public:
  ... ...
  void select（）;
 private:
  std::vector<IRITypeUnion_T> iri_vec_;
  std::vector<std::string> var_vec_;
  ... ...
  std::vector<IRITypeUnion_T> select_spo_vec_[3];
  ... ...
  BitMap_T cur_valid_row_bm_index_;
  uint64_t cur_valid_row_bm_index_cardinality_;
```

```
    ... ...
    std::shared_ptr<SharedQueryData> p_shared_;
    bool is_valid_;
};
```

每个 TripleQuery 代表了一个查询单元。iri_vec_和 var_vec_分别用来存储非类型变量和类型变量。select_spo_vec_用来存储该 TripleQuery 所对应的 Variable 变量的绑定值。这里以图 4-1 所示的数据为例。

Subject	Predicate	Object
a	q	b
b	l	x
a	m	v
c	w	d
z	n	a
b	s	m
s	a	u
k	b	a

图 4-1 RDF 数据实例

Figure4-1 RDF data instance

比如，当面对这样一条 sparql 语句时：

```
select ?pre ?obj where{
  "a" ?pre ?obj .
}
```

符合条件的数据为第一行和第三行，因此 select_spo_vec_[1] ={"q"，"m"}，select_spo_vec_[2] = {"b"，"v"}（坐标 1，2 代表了?pre 和?obj 在三元组中的位置，因为坐标 0 为非 Variable 类型变量，没必要保存数据）cur_valid_row_bm_index_={10100000}（未压缩表示），cur_valid_row_bm_index_cardinality_=2（表示该 cur_valid_row_bm_index_中置 1 元素数量）。

其中 p_shared_指向 SharedQueryData 类型变量。一般一个 sparql 语句包含多个查询单元，比如：

```
select ?name ?age ?country where{
  <rdfs://45ac92yhbf1109ac> <rdfs://type.object.name> ?name    .
  ?name <rdf://person.age> ?age
  ?name <rdf://person.country> ?country .
}
```

这些查询单元组成了一个查询集合（query set）。在进行查询时，这些查询集

合之间需要进行数据的合取或者析取操作（即求并集或者交集）。对于上面的 sparql 查询，对应的是三个查询单元，它们构成了一个查询集合，该查询集合的交集即为变量的查询的结果。

在构建查询集合的同时，会赋予每一个查询集合一个类型为 SharedQueryData 的数据结构。SharedQueryData 是方便彼此消息传递的一个共享变量区域。SharedQueryData 的结构定义如下：

```
class SharedQueryData{
 public:
  int make_cartesian_product_by_filter_vector_linked（
        std::string another_var_name，
        std::string var_name，
        std::vector<std::string>（&filter_var_val_vector）[2]）
  ... ...
  private:
   std::unordered_map<std::string，IRITypeUnionTag> var_val_type_;
   std::unordered_map<std::string，std::set<HV_T> hv_bound_vals;
   std::unordered_map<std::string，std::set<SS_T> ss_bound_vals;
   ... ...
   std::unodered_map<std::string，
    std::unordered_map<std::string，
     std::vector<std::string>> intermediate_result_;
   std::unodered_map<std::string，
    std::vector<std::string>> intermediate_result_col_name_;
 };
```

其中，var_val_type_记录了 Variable 对应的绑定值的数据类型（HashValue 或者 ShortString）；hv_bound_vals 和 ss_bound_vals 则记录 Variable 对应的绑定值集合。另外 SharedQueryData 还承担了最后将查询结果的图结构构建成矩阵表格形式的数据存储功能，因此 intermediate_result_和 intermediate_result_col_name_是用来实现此功能的。其中，intermediate_result_这个类型是一个擦除类型的图结构类型。因此，如图 4-2 所示，SharedQueryData 在本质上可以看成三组约束（TripleQuery）和一个变量集合（SharedQueryData）构成的结构体。

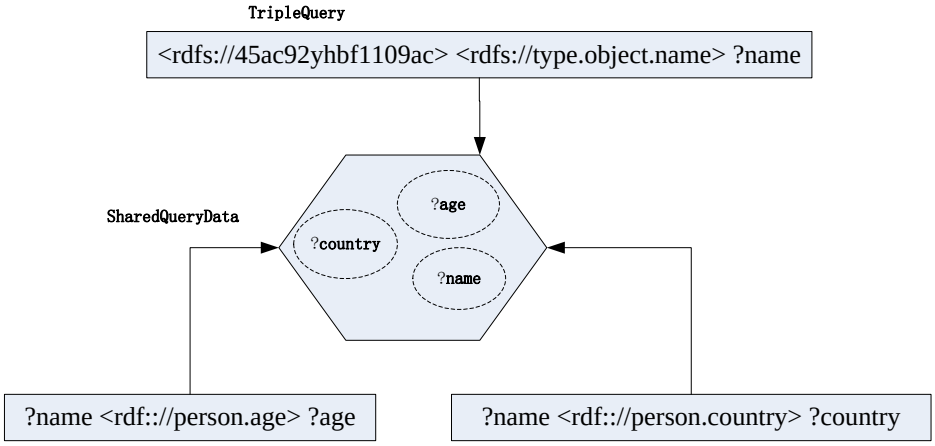


图 4-2 SPARQL 转化后的查询结构

Figure 4-2 the query structure which transformed from SPARQL

4.3 查询算法

根据 SPARQL 语句构建完约束集合（即 TripleQuery）后，需要将 TripleQuery 集合转化为一个查询图（query graph），按照这个查询图的顺序依次对约束进行求解，以最终获取满足所有约束的变量集合。

4.3.1 总体思路

一般的思路，对 TripleQuery 集合中的每一个查询单元执行查询，然后对查询结果的三元组进行取交集。但这样查询效率太低，会导致大量的硬盘 IO 和运行时开销。本设计采用 Bitmap 做索引是基于位图能够节省内存空间并方便进行逻辑运算，而且基于 Roaring 的压缩位图能够让数据在压缩的形态下执行高效的逻辑运算。因此可以将所有类似关系型数据库的连接（join）操作转换为逻辑与计算，并通过合理的优化方法逐步计算。

4.3.2 查询图构建与执行

查询分为以下步骤进行：

(1) 对每个 TripleQuery 进行存在型查询，计算位图数据 cur_valid_row_bm_index_和该位图非 0 值元素数量 cur_valid_row_bm_index_cardinality_。这一步利用非 Variable 的位图索引进行逻辑计算即可，不需要硬盘 IO ；

(2) 根据第一步存在性查询结果，确定执行的查询顺序。以此从查询集合中挑选出当前最佳的 TripleQuery，从硬盘获取对应的三元组，更新 QuerySharedData 中的 Variable 变量绑定值。这里的挑选标准为：1）cur_valid_row_bm_index_cardinality_ 较小的；2）与之前挑选出的 TripleQuery 中存在相同 Variable 变量的；

(3) 如果步骤 2 中获得约束结果集大小为 0，即 cur_valid_row_bm_index_cardinality_ 为 0，则直接进入步骤 4；否则重复进行步骤 2 的查询和计算，直到查询集合中的所有约束都被求解；

(4) 根据 QuerySharedData 中中最终获得的 Variable 变量集合，构建最终查询结果图，输出结果矩阵。

这里选取一个例子，来进行说明。RDF 数据如图 4-3 所示。

Subject	Predicate	Object
a	q	b
b	l	x
a	m	v
c	w	d
z	n	a
b	v	n
v	w	n
k	b	a
u	m	a
q	m	a
a	m	c

图 4-3 RDF 数据实例

Figure 4-3 RDF data instance

需要执行的 SPARQL 查询如下所示：

```
select ?var1 ?var2 ?var3 where {
  ?var1 "q" "b" .
  ?var1 "m" ?var2 .
  ?var2 "w" ?var3.
};
```

通过 SPARQL-Parser 解析之后，会生成如下三组约束和共享变量，如图 4-4，其中共享变量（var1，var2，var3）的绑定值未知。计算步骤如下：

(1) 首先对三组约束进行粗粒度的存在性查询：

1) 对第一组约束（?var1，"q"，"b"），根据数据获取逻辑，将关键词“q”的 pred 位图索引与关键词“b”的 obj 位图索引进行逻辑与操作，即 cur_valid_row_bm_index_，由 RDF 数据可知为（10000000000），即满足该约束的为第 1 行 RDF，因此 cur_valid_row_bm_index_cardinality_为 1；

2) 同理，计算第二组约束（?var1 “m” ?var2 ），可得该组约束的 cur_valid_row_bm_index_ 为（00100000111 ），cur_valid_row_bm_index_cardinality_ 为 4，满足该约束的存在 4 行数据；

3) 同理，计算第三组约束（?var2 “w” ?var3 ），可得该组约束的 cur_valid_row_bm_index_ 为（00010010000 ），cur_valid_row_bm_index_cardinality_ 为 2，满足该约束的存在 2 行数据；

注意，这一步中并没有进行任何磁盘读取的动作。

(2) 根据第一步的存在性查询结果确定查询图的执行逻辑。在这一步中将进行必要的磁盘读取，并对共享数据区的变量进行更新。根据 4.2.2 中的约束选取原则：

1) 选取满足约束数量少的 RDF（此时不存在以绑定值的共享变量），即第一组约束（?var1，“q”，“b”），由其 cur_valid_row_bm_index_可知该约束满足的数据为第一行，通过行坐标读取该三元组为（“a”，“q”，“b”），因此?var1 的绑定值集合更新为（“a”），将其在 SharedQueryData 中进行更新；

2) 因为第二组约束中存在具有绑定值的共享变量?var1，因此第二次计算选取第二组约束（?var1 “m” ?var2 ）。通过上一步的计算，我们更新的?var1 的绑定值集合，因此能够对该约束进行进一步的收缩（即条件过滤）。?var1 出现在第二组约束的主语位置，且现在的绑定值为“a”，我们获取术语“a”的主语位置的位图索引（即 10100000000）与当前该组约束的 cur_valid_row_bm_index_（即 00100000111）进行逻辑与操作，并更新到 cur_valid_row_bm_index_，则更新后的 cur_valid_row_bm_index_ 为（00100000000 ），cur_valid_row_bm_index_cardinality_ 同时更新为 1。由此，我们借助第一组约束的绑定值，将第二组约束的可能结果集从 4 收缩到 1。然后根据满足该约束的行坐标（即第三行）数据对硬盘读取，获取三元

组（“a”，“m”，“v”），可知?var2 的绑定值集合为（“v”）。

3) 继续选取，因为存在共享变量?var2，选取第三组约束集合（?var2 “w” ?var3），根据?var2 绑定值“v”的主语位置（00000010000）索引即当前该组约束的 cur_valid_row_bm_index_（00010010000）进行逻辑与操作，更新 cur_valid_row_bm_index_为 00000010000 ，进行硬盘读取，获取对应三元组(“v”,“w”,“n”)即?var3 的约束集合为(“n”)。

(3) 约束计算完成，SharedQueryData 中的变量：?var1 的绑定值为“a”，?var2 的绑定值为“v”，?var3 的绑定值为“n”。同时第二组约束中，存在绑定值映射 ?var2→?var3，即（“v” →“n”）。该绑定值映射关系将会在结果图构建过程中使用到。

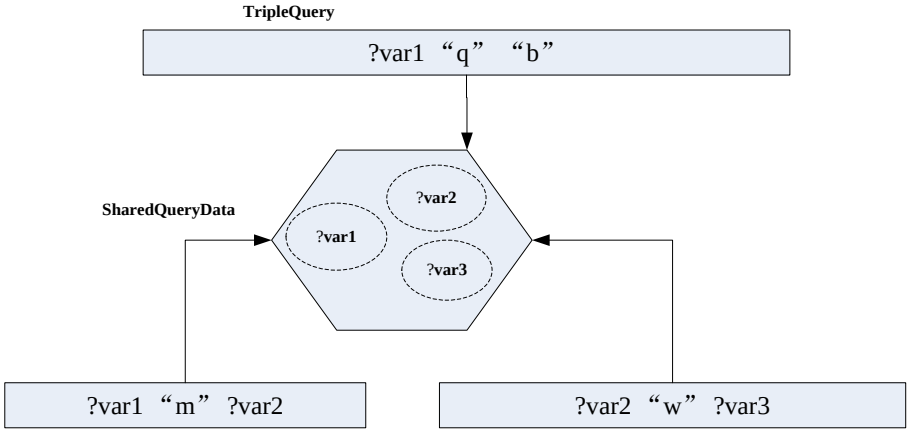


图 4-4 转换后的查询图

Figure 4-4 query graph transformed from SPARQL

4.3.3 结果图构建

通过查询图执行查询计算后，获得了符合 SPARQL 约束的变量的绑定值集合，和绑定值之间存在的映射关系，即获得了符合 SPARQL 约束的结果图。但一般情况下，客户端更希望这个结果图能够以二维表的形式返回。而结果图的构建主要由两步规约计算组成：

(1) 从图结构通过最小生成树算法转化树结构，这一步解决变量节点之间的循环依赖问题；

(2) 对树节点进行迭代计算笛卡尔乘积，这一步将树结构转换为二维表结构。

下面将详细介绍结果图的构建过程。

(1) “图”到“树”的规约

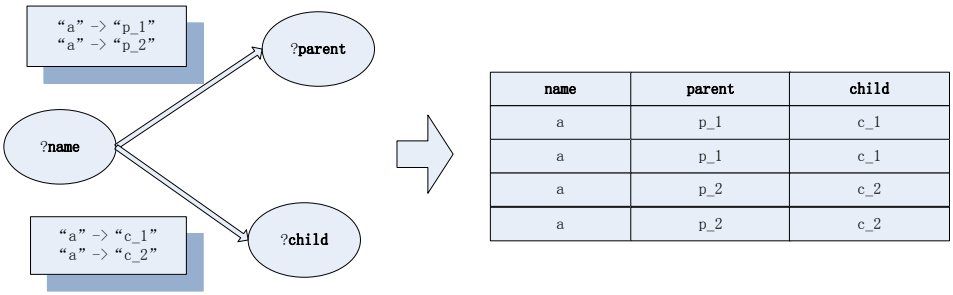
先来一个简单的例子，比如下面这条 SPARQL 查询语句：

```
select ?name ?child ?parent where{
    ?name <rdfs://person.profession> "doctor"    .
    ?name <rdf:://person.child> ?child .
    ?name <rdf:://person.parent> ?parent .
}
```

该查询的语句的语义可以理解为：查询职业为医生的人的名字及其孩子和父母的名字。假设经过基础查询计算后之后，获得了如下绑定的值及其约束关系：

1) ?name → ?parent : { "a" → "p_1" ; "a" → "p_2"}
2) ?name → child : {"a" → "c_1"，"a" → "c_2"}

如图 4-5 所示，以 Variable（即 name,parent,child）为节点，构建该图。



name	parent	child
a	p_1	c_1
a	p_1	c_1
a	p_2	c_2
a	p_2	c_2

图 4-5 绑定约束转化图

Figure 4-5 bound constraint transform table

通过将同一个绑定变量的值做笛卡尔乘积，即可获得对应的输出矩阵，如图 4-5 右侧所示。

图 4-5 只是简单的一种情况。更普遍的情况是，如图 4-6 所示，存在多个 Variable 节点，而且这些节点之间互相约束，构成有环的图。

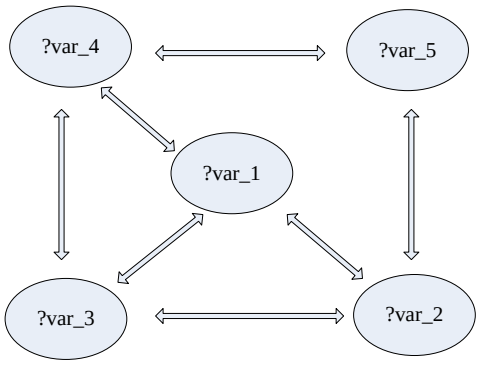


图 4-6 多值约束的有环图

Figure 4-6 cyclic graph which has mutil-value constraint

面对这种情况，需要把图提取出一颗树。而且，为了最小化之后的运算代价，把节点之间的“约束力”（即该条边对应的 TripleQuery 的成员变量值

cur_valid_row_bm_index_cardinality_）作为该边的权重。那么这个问题就规约为一个最小（权重）生成树问题。最小生成树定义如下：

一个给定的无向图 G=（V，E）中，V 是顶点的集合，E 是边的集合，（u，v）代表连接顶点 u 与顶点 v 的边（即（u，v）∈E），而 w（u，v）代表该边的权重，若存在 T 为 E 的子集（即 T∈V）且为无环图，使得

$$W(T) = \sum_{T \subseteq E} w(u, v)$$

最小，则 T 为 G 的最小生成树。如图 4-7，黑线所示路径即为该图的最小生成树。

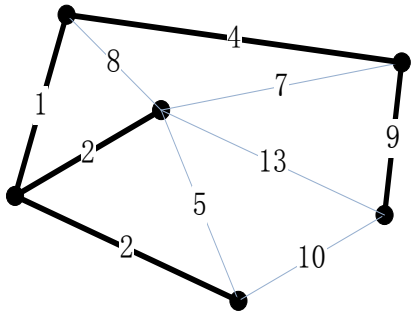


图 4-7 最小生成树

Figure 4-7 minimum spanning tree

最小生成树计算有很多算法，比较著名的是 Prim 算法和 Kruskal 算法。本系统实现采用 Kruskal 算法，该算法实现简单，而且对权重相同的边存在的情况处理的比较好。该算法实现非常直白：按照边的权重顺序（从小到大）将边加入生成树中，如果加入该边会使得该树形成环则不加入该边。最终这些边组成的就是该图的最小生成树。

算法描述如表 4-1 所示。

```
MST-KRUSKAL(G,w)
1:  A = Ø
2:  for each vertex v∈G.V
3:      MAKE-SET(v)
4:  sort the edges of G.E into nondecreasing order by weight w
5:  for each edge (u,v)∈G.E, taken in nondecreasing order by weight
6:  if FIND-SET(u) ≠FIND-SET(v)
7:      A = A∪{(u,v)}
8:      UNION(u,v)
9:  return A
```

表 4-1 Kruskal 算法

Table 4-1 Kruskal algorithm

2. “树”到“点”的规约

构建出最小生成树之后，需要一种算法，能够沿着这棵树的结构进行笛卡尔

积计算，最终构建出结果矩阵。

当前获得的最小生成树是一颗无根树（也就是没有指明根节点的树）。一种方案是，先将无根树转换为一块有根树（任意指明一个顶点即可），再从根节点开始，递归的计算笛卡尔乘积。如果当生成树中间节点较多时可能会导致整个递归栈的空间消耗过大。因此，本方案设计了一种节点合并（node-merge）的算法，实现了将递归转化为迭代计算，而且无需将无根树进行转化。

该算法描述如表 4-2 所示。

```
NODE_MERGE(G_mst)
1:  v_set := empty vertex set
2:  v2d_map := empty map which maps vertex to its degree
3:  for each E_uv ∈ G_mst.E:
4:      add u,v into v_set,
        and increase the degree of u and v by 1 in v2d_map
5:  while v_set not empty:
6:      for each vertex v in v_set:
7:          if v2d_map[v] != 1:
8:              continue
9:          calculate Cartesian product of v and v_l which
            is connected to v
10:         decrease the degree of v_l by 1 in v2d_map
            and remove v from v  set
```

表 4-2 节点合并算法

Table 4-2 node-merge algorithm

该算法首先统计该最小生成树的节点度值。然后进行迭代计算。迭代计算首先从节点集合中任一选择一个度值为 1 的节点 v_l，与跟它相连的节点 v_r 进行合并，即进行笛卡尔积运算。合并之后，将该节点 v_l 从节点集合去除，同时更新节点 v_r 的度值（因为合并了节点 v_l，因此对应的度值应该减 1）。继续迭代选取度值为 1 的节点进行 merge。直到节点集合最终剩下一个节点。该节点即为最终合并的节点。将该节点的绑定值矩阵即为最终的合并矩阵。

这里以一个例子直观解释一下，“树”到”点“的规约计算过程。假设通过查询图执行计算之后，获得如图 4-8 结构，其中大括号中是节点之间的绑定值映射关系。

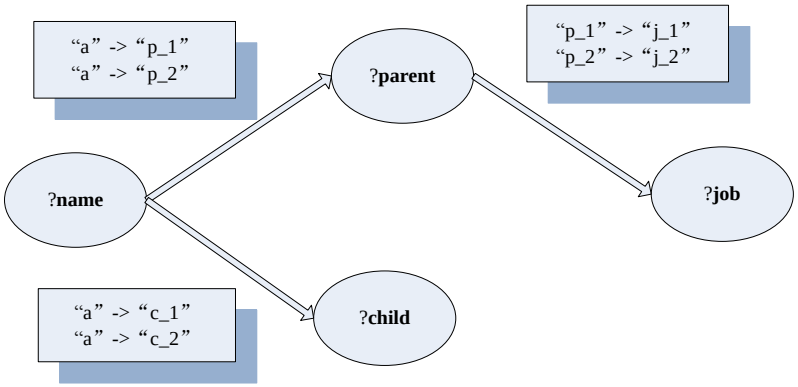


图 4-8 初始状态

Figure 4-8 initial state

首先选取度值为 1 的节点（?job，?child），这里以选取?job 节点，将其与它相连的节点进行合并，即进行笛卡尔积运算，如图 4-9 虚线框所示部分。

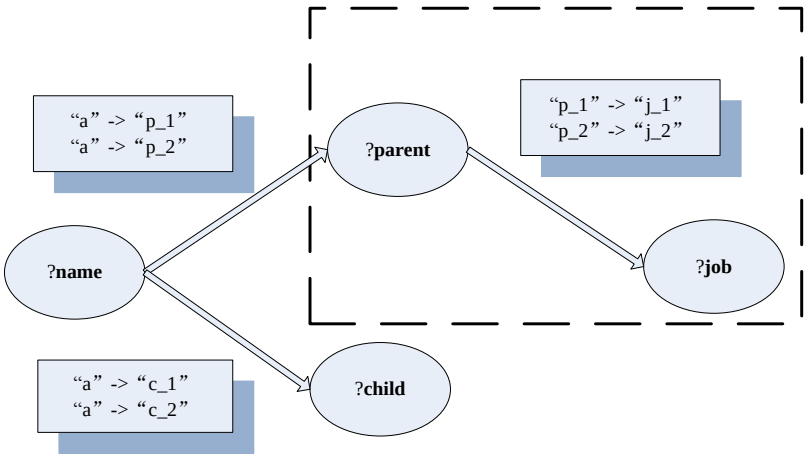


图 4-9 第一次合并

Figure 4-9 the first merging process

计算后，得到如图 4-10 结构，?parent 节点与?job 节点合并成一个新的节点：

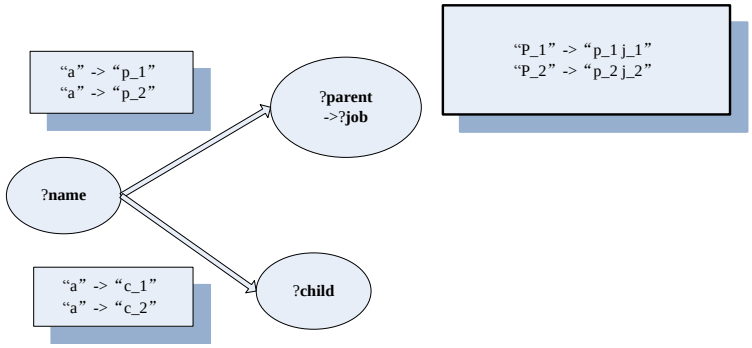


图 4-10 第一次合并结果

Figure 4-10 the status after the first merging

继续选取度值为 1 的节点，这里我们选取合并后的新节点，如图 4-11 虚线框

所示部分。

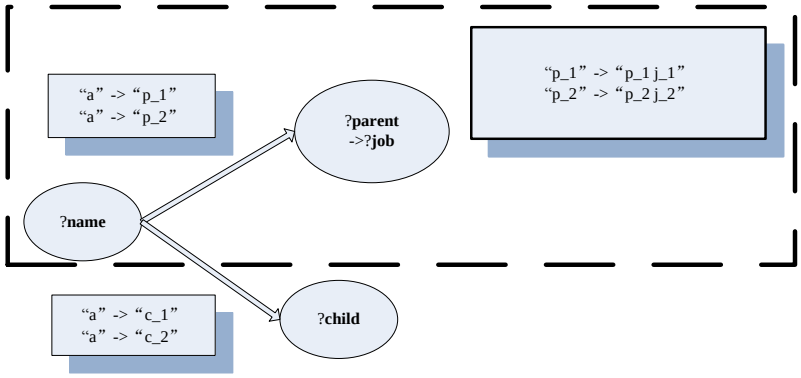


图 4-11 第二次合并

Figure 4-11 the second merging process

如图 4-12 所示，通过计算之后，得到一个新的合并节点。

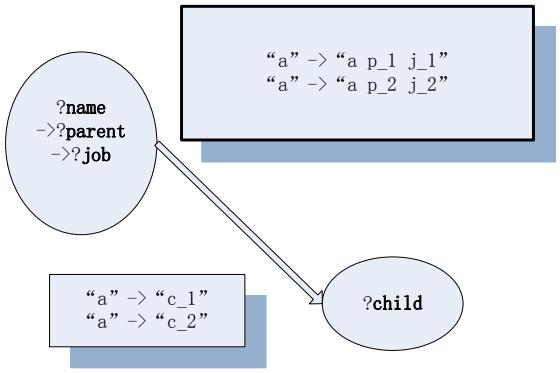


图 4-12 第二次合并结果

Figure 4-12 the status after the second merging process

继续选取度值为 1 的节点，进行合并，如图 4-13 所示。

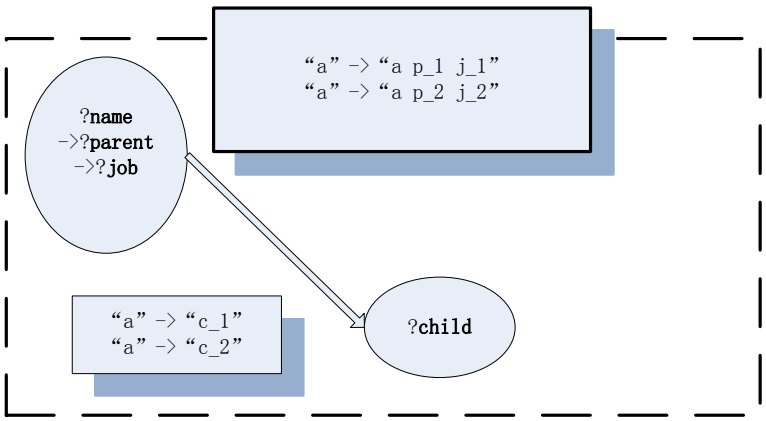


图 4-13 第三次合并

Figure 4-13 the third merging process

如图 4-14，直到最终树融合成一个节点，其映射结构如和映射数据如下。

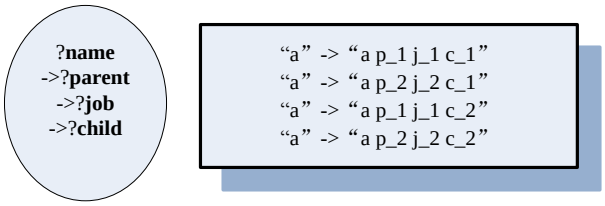


图 4-14 第三次合并结果

Figure 4-14 the status after the third merging process

因此，返回的二维表结构如图 4-15 所示。

name	parent	job	child
a	p_1	j_1	c_1
a	p_2	j_2	c_1
a	p_1	j_1	c_2
a	p_2	J_2	c_2

图 4-15 查询结果

Figure 4-15 query result

4.4 本章小结

本章主要介绍了 RDF 查询逻辑的设计和实现。简要说明了 SPARQL 解析逻辑。查询逻辑主要依靠位图索引高效的逻辑计算，将 SPARQL 查询语句转化为查询图，通过合理的查询图生成机制，进行高效的图约束计算。最后通过将“图”规约到“树”再规约到“点”的思路，进行笛卡尔积计算，最终获得二维表形式的查询结果集。

5 实验结果与分析

本章将对第三章提出的存储模型和第四章提出的查询算法进行实验评估。实验分别在单机伪分布式系统平台下对我们自己实现的图谱数据库 fishdb 利用测试数据集进行实验，实验对不同数据集分别执行查询，验证了存储模型和查询算法的可行性，并且与对 Google 开源的 Cayley 进行对比实验。最后对本章进行了小结。

5.1 实验环境

(1) 工程源码

设计实现的源码位于 https://github.com/WeeJang/fish-db.git，开源并遵循 GPL 协议。工程由 5000 行左右的 C++实现（不包含引用的其他库）。

(2) 编译环境

1) Clang-3.9
2) CMake（version :3.7.2）

(3) 实验环境

1) Linux kernel 3.13（x86_64）
2) Intel Core i5 @ 3.20GHz x 4
3) 6 GB 1600 MHz DDR3
4) SATA（500G）

(4) 测试数据集：

采用自己设计的学生关系数据集（已经在 github 上开源：https://github.com/WeeJang/kg_test_log.git），数据集参数如表 5-1 所示。

Item	**Size**
Row	6812270
Size	295M
Entity	3380000

表 5-1 测试数据集

Table 5-1 test data set

该数据集合图谱关系如图 5-1 所示。

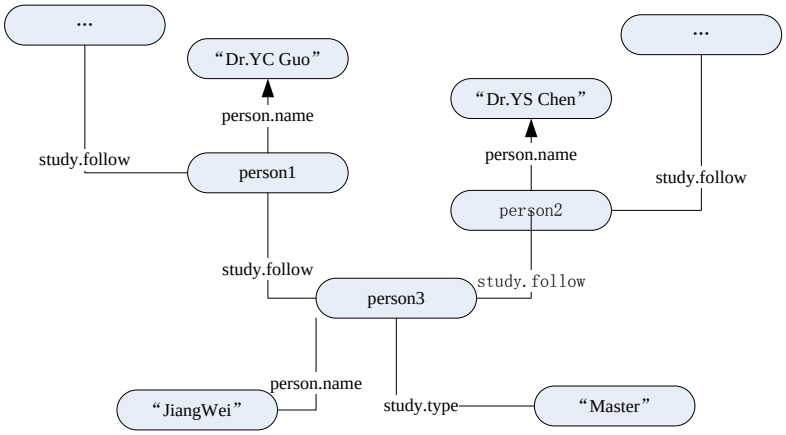


图 5-1 测试数据逻辑关系图

Figure 5-1 test dataset logical relation graph

5.2 数据加载

我们把该数据集分别格式化成 Cayley 和 fishdb 兼容的数据格式，然后进行加载。

Cayley 支持多种数据库后端的模式。如：直接基于内存的（memstore），或者基于第三方数据库（如 leveldb，mysql，mongodb，postgres 等）。在当前实验环境下，少量的数据集（<30M）可以采用直接基于内存的模式。这种模式的特点是，响应速度快，但是系统资源消耗太高。当我们采用自己的测试集进行测试时，这种模式基本没办法加载数据。因此只能选择以第三方数据库的模式。这里我们选用了性能比较高的键值型数据库 leveldb 作为 Cayley 的存储后端，leveldb 是基于 LSM 的存储引擎实现。

fishdb 无需多做配置，直接指定路径载入即可。数据加载时测量数据如表 5-2 所示。

	Cayley（leveldb）	fishdb
载入耗时（min）	36	7
峰值系统内存占用比（%）	42	70

表 5-2 数据载入对比

Table 5-2 data loading comparison

由表 5-2 可以看到 fishdb 的加载速度快于基于 leveldb 的 Cayley，但是峰值系统的资源消耗也大大高于 Cayley。

5.3 结果分析

我们对数据集进行 query 测试，我们选取一个典型的 SPARQL 测试用例：

```
select ?stu_name ?tec_name ?p_tec_name ?pp_tec_name where {
  ?stu_id <commlab://student.type> "master" .
  ?stu_id <commlab://person.name> ?stu_name .
  ?stu_id <commlab://study.follow> ?tec_id .
  ?tec_id <commlab://person.name> ?tec_name .
  ?tec_id <commlab://study.follow> ?p_tec_id .
  ?p_tec_id <commlab://person.name> ?p_tec_name .
  ?p_tec_id <commlab://study.follow> ?pp_tec_id .
  ?pp_tec_id <commlab://person.name> "Doc.X" .
}
```

这是一个典型的多结点知识图谱查询。上面这个查询的语义为“寻找硕士生的导师的导师的导师是 Doc.X 的人“。

使用该测试语句对 Cayley 和 fishdb 分别进行 15 次连续查询。

5.3.1 总体性能对比

将从查询耗时，内存资源消耗，CPU 利用率来对实验结果进行分析。

(1) 查询耗时对比

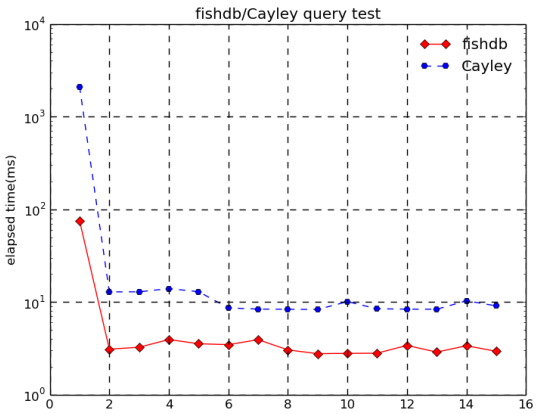


图 5-2 查询耗时对比

Figure 5-2 query time-consuming comparison

图 5-2 为全部 15 次测试的查询耗时图，两图的纵坐标分别为线性坐标和对数坐标。总体上来看，fishdb 的查询耗时是低于 Cayley 的。从图 5-2 中明显看到，经过第一次查询之后，两者的查询性能都有所提升，之后的查询性能趋于稳定。尤其是对于 Cayley，因为后端数据存储是基于 leveldb 的，系统冷启动时，绝大部分

数据是在硬盘中，没有载入内存或者系统的缓存（cache），因此第一次查询时，会发起大量的硬盘 IO，将数据读入内存，而且部分数据会被系统 cache 。当再一次发起查询时，由于绝大部分数据已经在内存中，因此之后的查询性能会提升 1.5 个数量级。对于 fishdb，同样如此，虽然 fishdb 中的位图索引在启动都都载入内存中，但是三元组的数据还是在硬盘里，因此经过一次查询之后，这些数据也会在操作系统层面被缓存，之后的查询也呈现出 0.5 个数量级的性能提升。

(2) 内存资源消耗对比

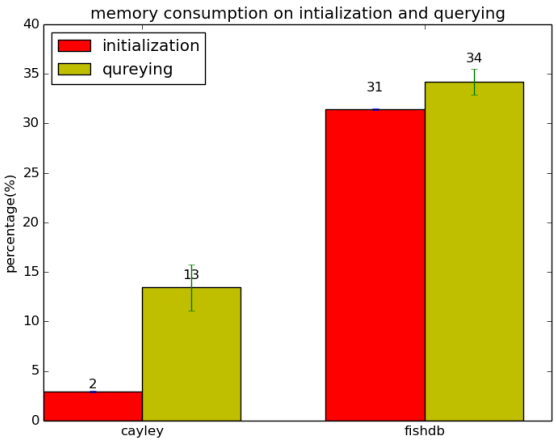


图 5-3 内存资源消耗对比

Fugure 5-3 memory consumption comparison

从图 5-3 可以看出，在系统在初始化状态时，fishdb 的内存消耗是远高于 Cayley 的。这是因为 fishdb 中在初始化时便将全部的位图索引都载入内存导致。在查询时，fishdb 的内存消耗增量时远小于 Cayley 的，这是因为在进行查询计算时，Cayley 需要读取大量的硬盘数据，并在内存中构建相应的数据结构，包括一些中间计算结果等。而 fishdb 因为在查询和计算时，主要依赖压缩的位图索引，包括优化的查询算法，使得内存在查询计算期间的消耗增量较小。

(3) CPU 利用率对比

从图 5-4 可以看出，相比较 Cayley 而言，fishdb 在查询计算期间的 CPU 利用率要提高了近 25%。这表明了 Cayley 比 fishdb 在查询计算期间有更多的硬盘 IO，影响了 CPU 的利用率。

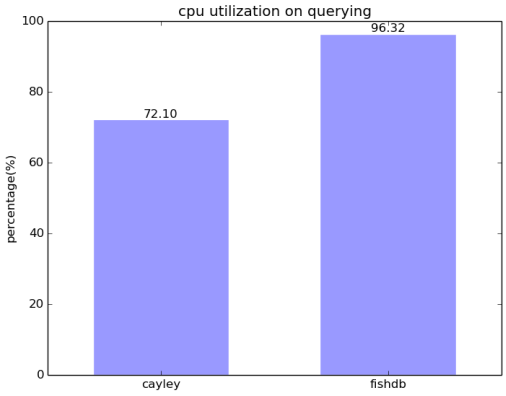


图 5-4 CPU 利用率对比

Fugure 5-4 CPU utilization comparison

5.3.2 在 no-cache 情况下的性能对比

数据的第一次查询，表现出了基于 leveldb 的 Cayley 和 fishdb 在没有 cache 情况下的性能表现，即冷启动情况下。如图 5-5，可以看到 fishdb 在冷启动的情况下，性能表现远远高于 Cayley.这是因为 fishdb 的位图索引基本都在内存中，在查询计算时直接利用位图进行逻辑计算和少量的硬盘 IO；而基于 leveldb 的 Cayley 需要通过大量的硬盘 IO 来读取数据。在这种情况下，更多的体现的是位图索引的优势。

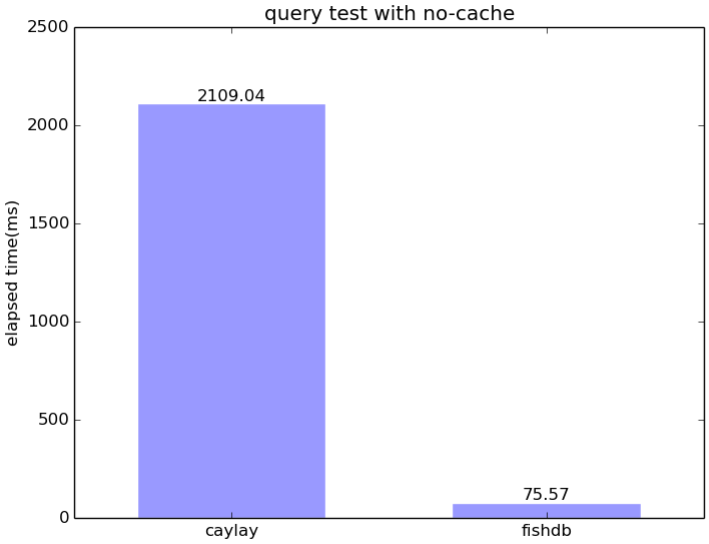


图 5-5 冷启动情况下查询耗时对比

Figure 5-5 query time-consuming comparison without caching

5.3.3 在 cache 情况下的性能对比

经过第一次查询之后，查询中的所需数据基本被缓存，因此不会出现由于 cache-miss 导致的硬盘 IO。这种情况下，我们可以对 cache 情况下的性能进行对比。我们选取了后 14 次查询数据，绘制其盒图。如图 5-6 所示。

可以看到，基于 leveldb 的 Cayley 的在缓存预热之后，耗时的均值在 9ms 左右，75%的耗时时间在 8.3ms～10ms 左右;而 fishdb 的性能明显优于 Cayley，耗时的均值在 3.3ms 左右，75%耗时时间在 3.1ms～3.5ms 左右，较 Cayley 而言更加稳定。在这种情况下，更多的体现的是查询算法的优势。

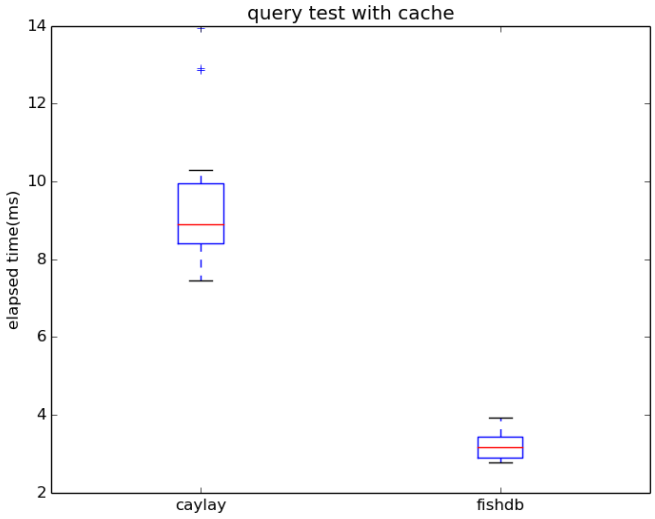


图 5-6 缓存预热后情况下查询耗时对比

Figure 5-6 query time-consuming comparison with caching

总体上看，fishdb 要优于基于 leveldb 的 Cayley，无论是在冷启动阶段，还是在缓存预热之后。在冷启动阶段，由于 fishdb 位图索引的优势，大大减少了硬盘 IO；在缓存预热之后，由于 fishdb 的查询算法，也能够避免不必要的硬盘 IO。使得总体性能带来提升，稳定性也更强。

5.3.4 对比总结

从上面的对比可以看出，在冷启动和缓存预热的情况下，fishdb 的查询性能都要优于 Cayley。而且在查询计算期间，fishdb 的 CPU 利用率和内存消耗增量两个指标也是优于 Cayley 的。但 fishdb 的总体内存资源占用要高于 Cayley，这也是使时间与空间的一种 trade-off。

这里需要评估一下这个权衡的代价：

(1) 在 no-cache 情况下，fishdb 相比较 Cayley 而言拥有 1.5 个数量级的响应性能提升，付出的代价是在初始化时需要多占用 0.9 个数量级的内存消耗；

(2) 在 cache 情况下，fishdb 查询计算时需要消耗 3%的增量内存空间，而 Cayley 则需要多消耗 11%的增量内存空间。如果在不考虑初始内存占用的情况下，假设两者的响应速度相同，则 fishdb 系统的吞吐量将是 Cayley 的近 3 倍；同时通过测试可以看到，fishdb 在 cache 情况下还具有 0.5 个数量级的响应速度优势，因此系统吞吐量的优势更为明显。

综上，在内存资源较为充足的情况下，这种 trade-off 是值得的。尤其是在一些对实时性要求较强的应用场景下。fishdb 能够在冷启动的情况下，依靠位图索引的优势，尽可能的减少不必要的硬盘 IO，降低了缓存不命中导致的系统性能波动，提高了系统的稳定性。

5.4 本章小结

本章采用测试数据集在单机伪分布式环境对基于本文存储方案的系统实现 fishdb 进行了性能测试，验证了存储模型和查询算法的可行性。并且与现有的开源 RDF 存储方案 Cayley 在查询性能和资源消耗等方面进行了比较。通过性能测试和结果分析可以看到，fishdb 在牺牲一定内存空间为代价的基础上，获得了较高的查询性能提升，验证了方案的有效性。

6 结论

随着 RDF 在各个领域的广泛应用以及数据量的急速增长，如何高效地管理海量数据是近年来的研究热点。现有的数据管理系统大都采用传统的关系型数据库来存储数据，这种方式已难以高效地管理海量数据。如何设计一种高效的，可扩展性的 RDF 数据存储和查询系统来解决海量数据管理问题具有重要意义。

论文主要围绕 RDF 数据的存储和查询设计展开。在索引上，使用了 Bitmap 作为索引的数据结构，并使用 Roaring 进行压缩，节省了运行时内存空间的使用；在存储上，使用了类似 HDFS 的分块存储技术，后期通过添加一致性组件可实现分布式存储。并通过实验验证这种方案的可行性与有效性。

本文的研究工作主要包括以下几个方面：

(1) 总结当前 RDF 数据管理方面的解决方案及相关技术。相关技术总结了当前比较主流且成熟的单机存储技术和分布式存储技术，以及基于这些存储技术的 RDF 存储管理方案；

(2) 提出了一种基于可扩展为分布式的数据存储模型。将 RDF 行数据文件按照块（chunk）存储，将每个块利用元数据索引进行组织和管理，最大程度的降低了硬盘存储开销，并提高了数据的容错性。

(3) 利用实现该存储模型，基于压缩的位图索引实现了的查询算法。该算法能够充分利用位图索引逻辑计算的优势，保证了高效的查询性能基础上；同时，使用可压缩的数据结构，降低了运行时内存资源的开销；

(4) 采用测试数据集在单机伪分布式系统对存储模型与查询算法进行了实验评估，验证了存储模型和查询算法的可行性，并且与 Google Cayley 的存储模型与查询算法进行比较，通过对比实验证明了本文提出的方案的有效性。

本文对数据存储与查询研究进行了初步的研究，还有很多的不足之处；在今后的工作中，还需要对以下几个问题做进一步的研究：

(1) 本文主要是围绕 RDF 三元组的数据实体的存储展开工作，并未将数据的类型信息加入（即 RDF Schema）。这一部分的内容也是 RDF 设计中的一个重要部分；

(2) 本文的 SPARQL 解析模块仅实现了基本的图模式查询，对于更多的 SPARQL 查询模式并未实现，如组图模式，可选图模式等。

参考文献

[1] Shadbolt N, Hall W, Berners-Lee T. The Semantic Web Revisited[J]. Intelligent Systems IEEE, 2006, 21(3):96-101.

[2] Klyne, Graham, Carroll, et al. Resource Description Framework (RDF): Concepts and Abstract Syntax[J]. World Wide Web Consortium Recommendation, 2004.

[3] Auer S, Bryl V, Tramp S. Linked Open Data -- Creating Knowledge Out of Interlinked Data[J]. Semantic Web, 2014, 1(1,2):97-104.

[4] Neumann T, Weikum G. The RDF-3X engine for scalable management of RDF data[J]. The VLDB Journal, 2010, 19(1):91-113.

[5] Carroll, Jeremy J, Dickinson, et al. Jena: implementing the semantic web recommendations[J]. 2004.

[6] Prud'Hommeaux E, Seaborne A. SPARQL Query Language for RDF[J]. 2007, 4.

[7] Widenius M, Axmark D P. Mysql Reference Manual[J]. Dec 2009 - World Bank, Washington, 2002(4).

[8] O'Neil P, Cheng E, Gawlick D, et al. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 1996, 33(4):351-385.

[9] Chan C Y, Ioannidis Y E. Bitmap index design and evaluation[J]. Acm Sigmod Record, 1999, 27(2):355-366.

[10] Wu K, Shoshani A, Otoo E. Word aligned hybrid bitmap compression method, data structure, and apparatus: US, US20040090351[P]. 2004.

[11] Chambi S, Lemire D, Kaser O, et al. Better bitmap performance with Roaring bitmaps[J]. Software Practice & Experience, 2014, 46(5):709-719.

[12] Brewer E. A certain freedom: thoughts on the CAP theorem[J]. 2010.

[13] Decandia G, Hastorun D, Jampani M, et al. Dynamo: amazon's highly available key-value store[C]// ACM Sigops Symposium on Operating Systems Principles. ACM, 2007:205-220.

[14] Chang F, Dean J, Ghemawat S, et al. Bigtable: A Distributed Storage System for Structured Data[J]. ACM Transactions on Computer Systems (TOCS), 2008, 26(2):4.

[15] Karger D, Lehman E, Leighton T, et al. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web.[C]// Twenty-Ninth ACM Symposium on Theory of Computing. ACM, 2001:654-663.

[16] Ghemawat S, Gobioff H, Leung S T. The Google file system[J]. Acm Sigops Operating Systems Review, 2003, 37(5):29-43.

[17] Rohloff K, Schantz R E. High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store[C]// Splash Workshop on Programming Support Innovations for Emerging Distributed Applications. DBLP, 2010:4.

[18] Myung J, Yeon J, Lee S G. SPARQL basic graph pattern processing with iterative MapReduce[C]// The Workshop on Massive Data Analytics on the Cloud. ACM, 2010:1-6.

[19] Papailiou N, Konstantinou I, Tsoumakos D, et al. H2RDF: adaptive query processing on RDF data in the cloud[C]// International Conference on World Wide Web. ACM,

2012:397-400.

[20] Punnoose R, Crainiceanu A, Rapp D. Rya:a scalable RDF triple store for the clouds[C]// Proceedings of the 1st International Workshop on Cloud Intelligence. ACM, 2012:1-8.

[21] George L. HBase - The Definitive Guide: Random Access to Your Planet-Size Data.[M]. DBLP, 2011.

[22] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[C]// Conference on Symposium on Opearting Systems Design & Implementation. USENIX Association, 2004:10-10.

[23] Neumann T, Weikum G. x-RDF-3X: fast querying, high update rates, and consistency for RDF databases[J]. Proceedings of the Vldb Endowment, 2010, 3(1-2):256-263.

[24] Weiss C, Karras P, Bernstein A. Hexastore: sextuple indexing for semantic web data management[J]. Proceedings of the Vldb Endowment, 2008, 1(1):1008-1019.

[25] Harris S, Gibbins N. 3store: Efficient Bulk RDF Storage[J]. In Proceedings of the First International Workshop on Practical and Scalable Semantic Systems, 2003, http://km.aifb.uni-karlsruhe.de/ws/psss03/proceedings/harris-et-al.pdf, 2003:1-15.

[26] Heo S Y, Kim E G. A Study on the Improvement of Query Processing Performance of OWL Data Based on Jena2[C]// International Conference on Convergence and Hybrid Information Technology. IEEE Xplore, 2008:678-681.

[27] White T. Hadoop: The Definitive Guide[M]. O'Reilly Media, Inc. 2010.

[28] Abraham J, Brazier P, Chebotko A, et al. Distributed Storage and Querying Techniques for a Semantic Web of Scientific Workflow Provenance.[C]// IEEE International Conference on Services Computing, SCC 2010, Miami, Florida, Usa, July. DBLP, 2010:178-185.

[29] Hauwirth M, Wylot M, Grund M, et al. Non-native RDF Storage Engines[M]// Handbook of Big Data Technologies. Springer International Publishing, 2017.

[30] 金强. 基于 HBase 的 RDF 存储系统的研究与设计[D]. 浙江大学, 2011.

[31] 朱敏. 基于 HBase 的 RDF 数据存储与查询研究[D]. 南京大学, 2013.

[32] 威滕, 莫法特, 贝尔梁斌,等. 管理海量数据 : 压缩、索引和查询 : Managing gigabytes : compressing and indexing documents and images : 经典再现全新修订版[M]. 电子工业出版社, 2014.

作者简历及攻读硕士/博士学位期间取得的研究成果

一、作者简历

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作和取得的研究成果，除了文中特别加以标注和致谢之处外，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交通大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

学位论文作者签名： 签字日期： 年 月 日

学位论文数据集

表 1.1： 数据集页

<table>
<tr><td>关键词*</td><td>密级*</td><td>中图分类号</td><td>UDC</td><td>论文资助</td></tr>
<tr><td></td><td>公开</td><td></td><td></td><td></td></tr>
<tr><td colspan="2">学位授予单位名称*</td><td>学位授予单位代码*</td><td>学位类别*</td><td>学位级别*</td></tr>
<tr><td colspan="2">北京交通大学</td><td>10004</td><td>工学</td><td>硕士</td></tr>
<tr><td colspan="2">论文题名*</td><td colspan="2">并列题名</td><td>论文语种*</td></tr>
<tr><td colspan="2"></td><td colspan="2"></td><td>中文</td></tr>
<tr><td>作者姓名*</td><td colspan="2"></td><td>学号*</td><td></td></tr>
<tr><td colspan="2">培养单位名称*</td><td>培养单位代码*</td><td>培养单位地址</td><td>邮编</td></tr>
<tr><td colspan="2">北京交通大学</td><td>10004</td><td>北京市海淀区西直门外上园村 3 号</td><td>100044</td></tr>
<tr><td colspan="2">工程领域*</td><td>研究方向*</td><td>学制*</td><td>学位授予年*</td></tr>
<tr><td colspan="2"></td><td></td><td>2</td><td></td></tr>
<tr><td>论文提交日期*</td><td colspan="4"></td></tr>
<tr><td>导师姓名*</td><td colspan="2"></td><td>职称*</td><td></td></tr>
<tr><td>评阅人</td><td colspan="2">答辩委员会主席*</td><td colspan="2">答辩委员会成员</td></tr>
<tr><td></td><td colspan="2"></td><td colspan="2"></td></tr>
<tr><td colspan="5">电子版论文提交格式 文本（ ） 图像（ ） 视频（ ） 音频（ ） 多媒体（ ） 其他（ ）
推荐格式：application/msword；application/pdf</td></tr>
<tr><td colspan="2">电子版论文出版（发布）者</td><td>电子版论文出版（发布）地</td><td colspan="2">权限声明</td></tr>
<tr><td colspan="2"></td><td></td><td colspan="2"></td></tr>
<tr><td>论文总页数*</td><td colspan="4"></td></tr>
<tr><td colspan="5">共 33 项，其中带*为必填数据，为 21 项。</td></tr>
</table>