学校代码：10286
分 类 号：TP393
密　　级：公开
U　D　C：004.9
学　　号：

东南大學

SOUTHEAST UNIVERSITY

工程硕士学位论文

基于大数据平台的知识图谱存储访问系统的设计与实现

(学位论文形式：应用研究)

研 究 生 姓 名：_____

导 师 姓 名：_____

校外导师姓名：_____

申请学位类别 工程硕士　　学位授予单位 东南大学

工程领域名称 计算机技术　　论文答辩日期 _____

研 究 方 向 数据库与信息系统　　学位授予日期 _____

答辩委员会主席 _____　　评　阅　人 _____

2018 年 月 日

Design and Implementation of Knowledge Graph Storage Access System Based on Big Data Platform

A Dissertation Submitted to

Southeast University

For the Professional Degree of Master of Engineering

BY

Supervised by

Southeast University-Monash University Joint Graduate School

Southeast University

April 2018

东南大学学位论文独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得东南大学或其它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。

研究生签名：　　日期：

东南大学学位论文使用授权声明

东南大学、中国科学技术信息研究所、国家图书馆有权保留本人所送交学位论文的复印件和电子文档，可以采用影印、缩印或其他复制手段保存论文。本人电子文档的内容和纸质论文的内容相一致。除在保密期内的保密论文外，允许论文被查阅和借阅，可以公布（包括以电子信息形式刊登）论文的全部内容或中、英文摘要等部分内容。论文的公布（包括以电子信息形式刊登）授权东南大学研究生院办理。

研究生签名：　　导师签名：　　日期：

摘 要

应用实体以及关系的语义知识图谱于搜索、问答和分析等场景需要可扩展存储模式和分布并行查询。本文在 Big Table 模型下设计具有存储负载分布均衡、局部节点聚集存储特点的分布聚集存储模式；采用 Group-By 模式分布并行计算查询树的分布并行查询引擎。实验验证，本文设计的存储模式和查询引擎具有良好的水平扩展性。具体工作总结如下：

（1）分布聚集存储模式：基于 Big Table 模型对逐行存储的实体集合经随机前缀和预分区操作进行均匀分割和分布存储，实现负载均衡；同时，随机前缀也能够对同类型的实体均匀地分布到节点存储，并在单个节点上按实体类别聚集。

（2）分布并行查询引擎：基于分布聚集存储模式设计两种采用不同方案的分布并行查询引擎：MIQE（Memory Iteration Query Engine）和 IIQE（Inverted Index Query Engine）。MIQE 采用分布式内存迭代技术以过滤和连接操作并行查询在内存中以抽象集合表示的实体，IIQE 将倒排索引与协处理器结合在集群中以并行索引查询的方式查询实体。上述两种查询引擎都旨在通过减少磁盘 I/O 和并行查询的方式提升知识图谱系统的读性能，加快知识图谱查询速度。

（3）原型系统的实现和性能验证：基于上述研究，本文设计并实现基于大数据平台的知识图谱存储访问系统。实验验证，基于分布聚集存储模式和分布并行查询引擎的知识图谱存储访问系统具有良好的水平扩展性，面对大规模知识图谱查询，IIQE 查询引擎的查询性能更加优异。

关键词：知识图谱；存储；查询；水平扩展性；Big Table

Abstract

The Knowledge Graph applied entity and relationship needs scalable storage schema and distributed parallel queries in search, question and answer (Q&A) and analysis scenarios. In this paper, we design a distributed-aggregated storage schema based on Big Table model, which has the characteristic of load- balanced storage and local node clustered storage and we design a distributed parallel query engine based on Group-By mode by scanning query tree in parallel. The experiments show that the storage schema and query engines we design have good horizontal scalability.The specific work is summarized as follows:

(1) Distributed-aggregated storage schema. We perform random prefix and pre-partition operations on row-by-row stored entity sets based on Big Table model. After these operations, we divide and store the entities evenly to achieve load-balancd. The random prefix can also distribute the same type of entity evenly to the node storage and aggregate them by the entity category on a single node.

(2) Distributed parallel query engines: We design two distributed parallel query engines that use different schemes based on the distributed-aggregated storage model: Memory Iteration Query Engine (MIQE) and Inverted Index Query Engine (IIQE). MIQE uses a distributed memory iteration technique to query entities that are represented in memory abstract set based on filter and join operations. IIQE combines inverted indexes with coprocessors in the cluster to query entities in parallel indexed queries. Both of the above query engines we design can improve the reading performance of the Knowledge Graph by reducing disk I/O and parallel query methods.

(3) Prototype system implementation and performance verification: We design and implement a Knowledge Graph storage access system based on big data platform according to the above research. Experiments show that the Knowledge Graph storage access system based on distributed-aggregated storage schema and distributed parallel query engine has good horizontal scalability. In the face of large-scale Knowledge Graph query, IIQE engine has the better query performance than other engines.

Keywords: RDF; Storage; Query; scalability; Big Table

目录

第一章 绪论

1.1研究背景

Google 于 2012 年通过知识图谱实现语义匹配的搜索[1]，改变了传统的基于字符匹配的搜索方法。知识图谱是一种有向图结构，描述现实世界中存在的实体和概念以及它们之间的关系。搜索引擎可以利用知识图谱对查询关键词进行语义扩展，从而提高搜索质量。除了智能语义搜索，知识图谱还广泛应用于问答和情报分析的场景，用于提高问答的准确度和对情报做更深层次的分析和决策。为了对智能搜索、自动问答和情报分析等应用提供支撑，知识图谱必须具备对知识高效存储和查询的能力。

面对数量庞大且不断增加的实体和关系，知识图谱需要可扩展存储模式和分布并行查询来保证知识图谱存储查询性能。本文基于大数据平台提出分布聚集存储模式和分布并行查询引擎实现知识图谱存储查询的扩展。分布聚集存储模式的设计思路：基于 Big Table 模型对逻辑上的一张大表在物理上均匀分割分布存储，使得实体存储具有总体负载分布均衡、局部聚集存储的特点。分布并行查询引擎的设计思路：基于设计的分布聚集存储模式，分别采用分布式内存迭代技术和倒排索引技术设计并行查询引擎，旨在通过减少磁盘 I/O 和并行扫描查询树的方式提升知识图谱的查询性能。海量知识图谱存储查询的难点和本文的改进成果如下图 1-1 所示。如下左图所示，面对大规模数据，知识存储发生数据倾斜，数据并未完全存储到所有服务器中，若对某种类型的实体查询，则会发生数据热点问题，影响知识图谱查询性能。若以本文设计的存储模式和查询引擎存储查询知识图谱，如下右图所示，不仅可以保证系统存储负载均衡、节点存储聚集而且充分利用集群性能保证系统的查询效率，使知识图谱的存储查询具有良好的水平扩展性。

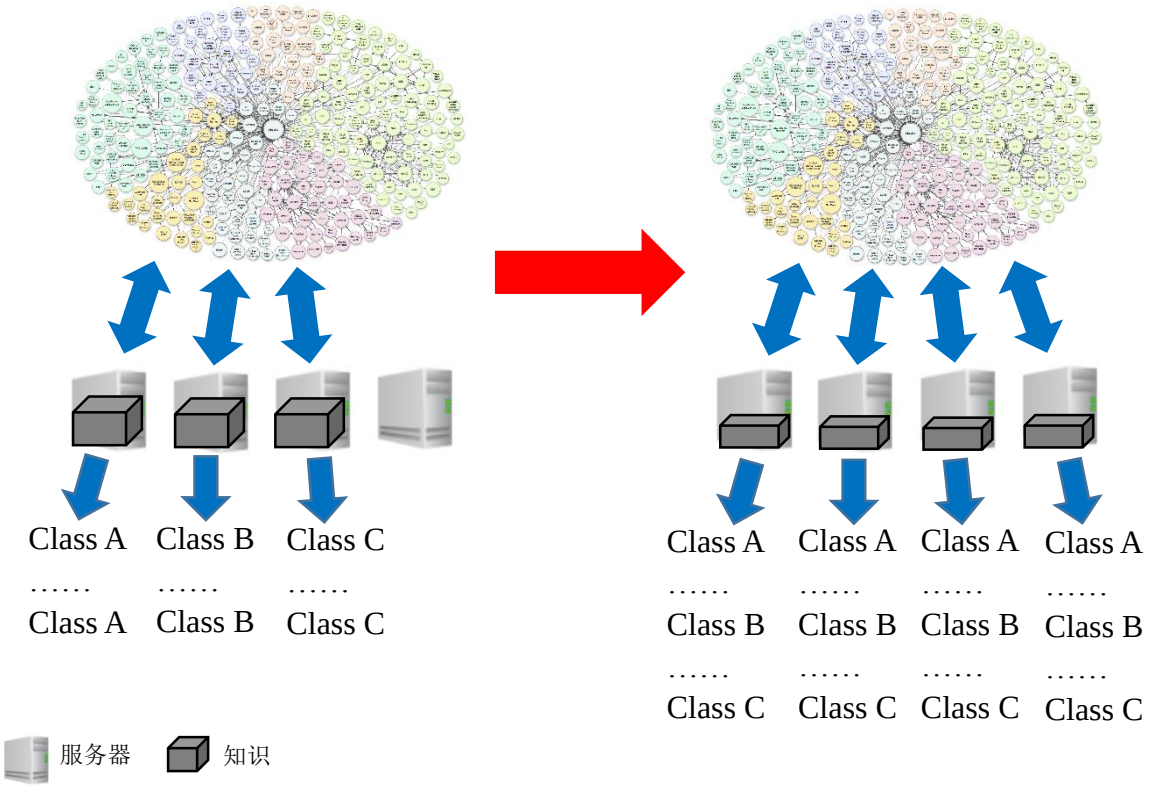


图 1-1 知识图谱存储查询难点和改进成果

1.2 研究现状

1.2.1 关系型数据库存储知识图谱

基于关系型数据库存储查询性能稳定、索引建立方式简单和易使用的特点，许多研究者提出使用关系型数据库的二维数据表对以三元组（主语，谓语，宾语）表示的知识图谱进行存储和查询，知识图谱基于表结构的存储模式常用的有 3 种：三元组表，属性表和垂直分割表。

论文[2]指出三元组表是用来存储知识图谱实体和关系最简单和最直接的方法，三元组表使用三个字段分别存储三元组的主语、谓语和宾语。三元组表以元组为单元进行存储，语义较为明确。使用三元组表存储模式不仅可以快速的存储知识图谱而且还可以在不需要修改表存储模式的情况下进行知识图谱本体推理的操作。但是三元组表会将整个知识图谱存储在一张表中，使单表规模过大，不利于查询、插入和修改等操作，同时面对复杂查询查询开销巨大，易降低查询的效率。

基于三元组表存储模式，有研究者[3]提出使用一种改进的属性表存储模式存储知识图谱，目前属性表存储模式已使用在 Jena Semantic Web toolkit[4]中。属性表存储模式是将三元组中谓语近似的主语汇聚成一张表存储。属性表中每一条记录表示一个实体，每一个字段表示该实体的一个属性或者关系。属性表存储模式相比于三元组表存储模式在同一个概念下实体间的一阶关系查询时性能较好，但当查询关系大于一阶且涉及到不同概念的实体时开销巨大，限制了知识图谱对复杂查询的处理能力，同时查询时属性表存储模式必须指定属性才能查询，限制了知识图谱的查询方式。

除了上述两种存储模式，还有论文[5]提出一种全分解存储模型存储三元组，即垂直分割存储模式。垂直分割存储模式将存有所有三元组的一张大表根据谓语划分成多张小表，其中三元组的谓语为表名，每张表只有主语和宾语两个字段。垂直分割存储模式不仅数据结构清晰易于理解而且不会存储 Null 值，但是垂直分割存储模式存在数据表过多，删除修改关系代价大等缺点。

上述三种基于关系型数据库的知识图谱存储查询方案都有各自的优缺点，实际使用时需要根据不同的应用场景进行选择，并没有绝对的最优和最差，再配合索引和约束技术可以加快查询速度，减少查询时间。Franke 等[6]使用 MySQL Cluster[7]以三元组表存储模式对 RDF（Resource Description Framework，资源描述框架）数据集进行存储和查询测试。经 LUBM[8]基准测试后发现，面对较小的数据集，基于三元组表存储模式存储的知识图谱存储性能优异，但是随着数据量的增加，当数据量达到 GB 时，系统的存储性能急剧下降，远不能达到小数据量时的存储速度。查询时，面对小数量集的知识图谱，在不涉及到 1 度以上的连接查询时，使用关系型数据存储格式可以保证知识图谱的查询性能但是当涉及到 2，3 度的连接查询，系统查询性能下降严重并且随着数据量的增加，系统查询性能急剧下降。实验说明使用关系型数据存储格式存储的知识图谱不利于存储查询的扩展。

1.2.2 非关系型数据库存储知识图谱

除了以基于表结构的关系型数据库存储知识图谱，由于知识图谱本质是一个图，现也常用图结构存储知识图谱。目前通用的以图结构存储知识图谱的方案有两种：RDF 存储和图数据库存储。有研究者常使用文档型数据库和分布式面向列的数据库存储 RDF 数据，另外也有研究者常使用图数据库存储知识图谱。

基于文档型数据库 MongoDB，论文[9]设计并实现基于 RDF 三元组的知识图谱存储访问系统。MongoDB[10]是目前最流行的文档数据库，它是一个基于分布式文件存储的高性能、开源的文档数据库，没有严格的数据模式，支持动态查询，全内容索引和自动故障处理等操作，有着高性能、可扩展、易部署的特点[11]。在文档型数据库中文档是数据存储的基本单位，数据都是以文档的方式存储[12]。根据文档型数据库的特点，该论文使用三张表存储 RDF 三元组，分别是 Resource 表，Litera 表和 Statement 表。Resource 表存储资源（resources），资源包括所有的实体、属性和关系。Litera 表存储字面量（literals），字面量存储各种类型的数值，如数字、字符串和日期等。Statement 表则以 Resource 表和 Litera 表对应的 ID 存储 RDF 三元组语句。三张表通过 id 之间的映射避免数据冗余。存储时，该论文使用 Jena[13]解析 RDF 文件将 RDF 三元组存储到相应的表中。查询时，该论文基于 MongoDB 的全内容索引技术建立索引提高系统查询效率。但是该论文只在单节点的小规模数据集下进行实验，实验过于单薄，并没有在大规模数据集下进行多节点的查询性能测试，同时以建立索引的 Mongo 与未建立索引的分布式面向列的非关系数据库 HBase[14]比较查询性能也有所不公。

分布式面向列的非关系型数据库的数据存储主要特点有[15]：1. 列式存储。所有的数据是基于列存储而不是关系型数据库中的行存储。2. 稀疏。对于列数据为空的情况下并不占用存储空间。3. 极易扩展。分布式面向列的非关系型数据库具有良好的水平扩展性，水平添加机器即可提升数据存储和查询能力。目前，HBase 是基于 Hadoop[16]技术的分布式的面向列的非关系型开源数据库的代表。论文[17]基于分布式面向列的非关系数据库 HBase 提出对 RDF 三元组每个谓语建立两张表（SO，OS）的存储方案。但是该方案面对大量拥有不同谓语的 RDF 三元组集合时会生成较多表，不利于数据管理且多表存储易造成数据的不一致。Franke 等[6]基于 HBase 提出总共使用两张表（SPO，OPS）的多列存储 RDF 三元组的存储方案并与基于三元组表存储模式存储知识图谱的关系型数据库 MySQL Cluster 进行查询性能的比较，实验验证，面对大规模 RDF 数据集时，Franke 等[6]方案的查询性能较为优异。Franke 等[6]方案相比于 Li[7]方案减少了表的个数，利于管理，但是仍易造成数据冗余同时还会造成数据热点问题。

除了使用 RDF 存储知识图谱，图数据库存储访问知识图谱也是一个非常热门的研究方向。论文[18]指出现使用图数据库存储知识图谱中可以解决知识图谱多阶连接操作所造成的性能下降问题。图数据库一般包含节点、边、属性这三个元素。存储知识图谱时，节点存储实体，边可以存储表示实体之间的关系，属性存储知识图谱的属性和字面量。图数据库不仅在多个连接操作上查询效率高，而且其存储模式设计灵活，在添加新的知识时，一般只需局部改动部分节点即可，改动代价小。有研究者[19]已使用图数据库 Neo4j[20]建立了中文商业知识图谱，该研究者成功地将实体和实体的关系存储在 Neo4j 便于知识图谱的绘制和查询。Neo4j 是当下稳定、成熟且具有较高性能的开源图数据库，具有优秀的读写性能。该论文中的实验验证基于 Neo4j 构建的中文商业知识图谱的存储查询性能优异，但是由于该论文建立的中文商业知识图谱数据量小，因此并未说明 Neo4j 在大规模数据下知识图谱存储查询的扩展性能。

1.3 研究内容

本文主要研究知识图谱存储查询性能的水平扩展性问题。本文通过对知识图谱存储访问系统设计可扩展存储模式和并行查询引擎来满足大规模知识图谱存储查询的性能需求，主要工作内容如下：

（1）分布聚集存储模式的设计。为了保证大规模知识图谱的存储性能，本文基于 Big Table 模型稀疏、分布式、一致、多维排序的特性，设计出单表多列簇的分布聚集存储模式。该存储模式通过对逐行存储的实体集合经随机前缀和预分区操作进行均匀分割和分布存储，同时，随机前缀也能够对同类型的实体均匀地分布到节点存储，并在单个节点上按实体类别聚集。该存储模式不仅拥有结构清晰、数据不冗余的特点还保证知识图谱存储查询的负载均衡状态。

（2）分布并行查询引擎的设计。为了保证大规模知识图谱的查询性能，基于提出的分布聚集存储模式，本文设计出采用分布式内存迭代技术的查询引擎 MIQE（Memory Iteration Query Engine）和采用倒排索引和协处理器技术的查询引擎 IIQE（Inverted Index Query Engine）旨在通过减少磁盘 I/O 和并行查询的方式提升知识图谱系统的读性能，保证知识图谱查询速度。MIQE 利用 Spark 内存计算模型将存储的所有实体已抽象集合的形式加载到集群内存，然后以内存迭代的方式使用过滤和连接等操作并行查询符合查询条件的实体。IIQE 通过在行键中建立倒排索引来减少查询时磁盘的读写次数，同时借助协处理器并行扫描服务器端机器加快查询速度。

（3）原型系统的实现和性能验证。基于上述可扩展存储模式和分布并行查询引擎的研究，本文设计并实现知识图谱存储访问原型系统并对系统性能进行测试。原型系统主要分为三大模块：数据载入模块，SPARQL 查询语句解析模块和数据查询模块。知识首先根据设计的存储模式通过数据载入模块存储于 HBase。当用户发起查询时，系统会对用户输入的 SPARQL 查询语句通过设计的分布并行查询引擎进行查询并返回结果。实验验证，本文设计并实现的基于分布聚集存储模式和分布并行查询引擎的知识图谱存储访问系统具有良好的水平扩展性。面对大规模知识图谱查询，IIQE 查询引擎的查询性能更加优异。

1.4 论文组织结构

本文一共分为六章，各章节的主要内容安排如下：

第一章是全文的绪论。该部分主要描述知识图谱存储查询的研究背景及研究现状，最后描述本文的研究内容及并给出论文组织结构。

第二章是全文的相关技术与研究部分。该部分对本文涉及到的相关理论和技术进行说明，包括知识图谱存储的相关理论和目前大数据存储处理的相关技术。

第三章是知识图谱存储访问系统存储模式设计部分。该部分基于 Big Table 模型设计出满足存储负载均衡的分布聚集存储模式并对该存储模式进行存储性能的测试。

第四章是分布并行查询引擎的设计部分。基于分布聚集存储模式该部分设计了 MIQE 和 IIQE 两种并行查询引擎，这两种并行查询引擎旨在通过减少磁盘 I/O 和以并行查询的方式提升查询性能。

第五章是原型系统的实现和性能验证。该部分设计并实现知识图谱存储访问原型系统并对系统性能进行验证。

第六章是总结全文，讨论目前系统的优缺点，指出未来的研究方向。

第二章 相关技术与研究

由半结构化数据组成的知识图谱需要使用合适的描述方法对知识进行表示和存储，同时为了便于对知识的查询，知识图谱还需要类 SQL 的查询语言支持各种查询条件下知识的查询，除此之外，面对大规模知识，知识图谱需要使用性能优异的数据管理系统和数据处理技术来保证知识图谱的可用性。本章主要是对知识图谱存储查询的相关理论和大数据存储处理的相关技术进行说明。知识图谱存储查询的相关理论主要说明知识图谱的数据模型、查询语言和查询类型。大数据存储处理技术主要描述目前广泛使用于知识图谱存储的文档型数据库 MongoDB、图数据库 Neo4j 和分布式面向列的数据库 HBase 的优缺点并对现流行的分布式内存计算引擎 Spark 进行相应介绍。最后对本章内容进行小结。

2.1 知识图谱存储查询的相关理论

知识图谱是基于 Berners-Lee 等[21]提出的语义网的研究，本质上是一种揭示实体之间关系的语义网络，用于表示实体以及实体间的二元关系。接下来本节将会介绍知识图谱的数据模型和查询语言。

2.1.1 知识图谱数据模型

知识图谱是一个结构化的语义知识库，它以符号形式描述真实世界中存在的各种实体，概念和关系。每个实体或概念用一个全局唯一确定的 ID 也称标识符来标识。每个属性-属性值对用来刻画实体的内在特性，关系则用于连接两个实体刻画实体之间的关联。知识图谱是一种基于图的数据结构，目前图结构存储知识图谱有两种通用的存储方案：RDF 存储和图数据库（Graph Database）。本小节主要介绍 RDF 存储，图数据库存储将在 2.2.2 节进行介绍。

RDF 是一种使用精确规整的词汇表来表达主张的断言语言，它既可以在 Web 上交互数据还可以保留数据的原有含义[22]。W3C（World Wide Web Consortium，万维网联盟）将 RDF 作为描述语义 web 上信息资源的标准模型[23]。RDF 使用主谓宾结构的三元组描述任何事物，如下图 2-1 所示，下图以主语：semantic-web-book，谓语：publishedBy，宾语：crcprocess）的方式描述 semantic-web-book——publishedBy——crcprocess 这个事物。

图 2-1 描述书与 CRC 出版社的关系的 RDF 三元组

RDF 图是直观表达 RDF 三元组的一种方式，它由一系列的 RDF 三元组组成。希茨利尔[24]在书中介绍到：RDF 图是一个基于图的数据模型，它提供了一个描述实体的统一

标准。一个 RDF 图由一个以节点和有向边构成的有向图描述，RDF 图的结点和边都用 URI（Uniform Resource Identifiers，统一资源标识符）进行标记。若以 RDF 存储知识图谱，知识图谱将以主谓宾形式描述的 RDF 图与知识图谱中的“实体-属性-属性值”和“实体-关系-实体”联系起来，如下图 2-2。具体来说，RDF 图中的主语是一个被描述的实体并以唯一的 URI 表示，不同的实体拥有不同的 URI。RDF 图的谓语表示实体的属性或者实体与实体之间的关系。当谓语表示属性时，宾语为属性值，一般是一个字面量（字符串或者其他数值类型）。当谓语表示实体与实体之间的关系时，宾语为另一个被描述的实体或者是空白节点[25]。RDF 图与 RDF 三元组相似，但是当宾语为属性值时 RDF 图以矩形表示。

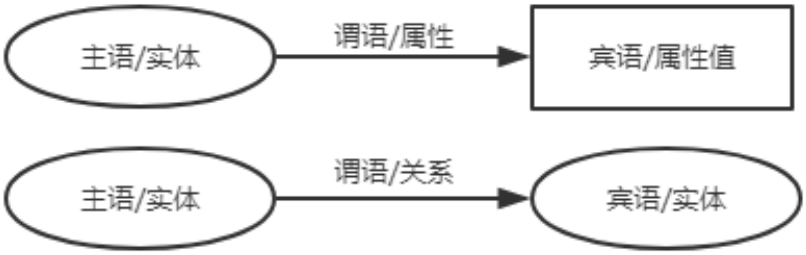


图 2-2 RDF 图表示的知识图谱

目前常以 RDF/XML，N-Triple 和 Turtle 等形式描述 RDF 图[26]。RDF/XML 是使用 XML 表示 RDF 数据，提出该方法主要是因为 XML 技术成熟，可以通过许多工具来使用 XML，但是基于 XML 格式冗长，不易阅读的特性，目前已不太使用 RDF/XML 方式处理 RDF 三元组。N-Triple 是 2004 年 W3C 提议的一个 RDF 可行语法，在 N-Triple 文件中，每一行表示一个三元组，开放领域知识图谱 DBpedia[27]就是使用 N-Triple 格式来表示 RDF 图。Turtle 是基于 RDF/XML 和 N-Triple 格式的改进，它比 RDF/XML 结构清晰且比 N-Triple 易读。结合图 2-1 的内容，下表 2-1，表 2-2 和表 2-3 给出相同内容的 RDF/XML，N-Triple 和 Turtle 三种具体表示。在表 2-1 的数据片段中，命名空间“http://www.w3.org/1999/02/22-rdf-syntax-ns#”被约定前缀为 rdf，命名空间“http://example.org/”被约定前缀为 ex，其余均是 XML 的标签名，RDF/XML 常以前缀+标签名的形式表示 URI，例如：“<ex:publishedBy>”表示“http://example.org/publishedBy”。表 2-2 使用 N-Triple 格式，一行表示一个 RDF 三元组。表 2-3 使用基于 N-Triple 的 Turtle 格式，对 RDF 的 URI 前缀进行缩写。

表 2-1 RDF/XML 表，描述书与 CRC 出版社的关系

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:ex="http://example.org/">
    <rdf:Description rdf:about="http://semantic-web-book.org/uri">
        <ex:publishedBy>
            <rdf:Description rdf:about="http://crcpress.com/uri">
            </rdf:Description>
        </ex:publishedBy>
    <rdf:Description>
</rdf:RDF>
```

表 2-2 N-Triple 表，描述书与 CRC 出版社的关系

<http://semantic-web-book.org/uri> <http://example.org/publishedBy> <http://crcpress.com/uri>.

表 2-3 Turtle 表，描述书与 CRC 出版社的关系

@prefix book: <http://semantic-web-book.org/> . @prefix ex: <http://example.org/> . @prefix crc: <http://crcpress.com/> . book:uri ex:publishedBy crc:uri .

2.1.2 知识图谱查询语言和查询类型

对以 RDF 图表示的知识图谱，常使用 SPARQL（Simple Protocol And RDF Query Language）查询语言[28]从 RDF 图中获取信息。SPARQL 查询语言与 SQL 语法相似，但本质不同，SQL 是专门针对于关系数据的查询语言而 SPARQL 查询语言是专门用于 RDF 的查询语言。SPARQL 查询语言的核心是以简单图模式为形式的简单查询，一般来说，简单图模式可以表示在知识图谱中要被查询的 RDF 图，因此 SPARQL 查询语句就是在以 RDF 三元组表示的 RDF 图中找到相应的 SPARQL 查询子图。简单图模式的基本语法大致与 Turtle 语法相同，不仅包括 Turtle 语法中关于逗号、分号的简化和对空白节点、RDF 集合的处理方法而且还引入查询变量作为表示查询返回结果。查询变量以符号?或者$作为首字符，接着是一串数字、字母和合法的特殊字符（比如下划线）。但是首字符?或者$并不是变量名的一部分。查询变量不仅可以作为主语或者宾语（即实体或者值）出现，还可以作为谓语（即属性或者关系）出现。SPARQL 除了支持简单图模式外还支持复杂的图模式，复杂的图模式由多个简单图模式组成。

知识图谱的查询一般分为三大类：单个实体查询，连接查询和实体关系查询。单个实体查询是指通过一个或多个{属性-属性值}和{关系-实体}对查找对应实体的查询。连接查询是指已知实体间关系查找对应实体的查询。实体关系查询是指对知识图谱实体之间未知关系的查询。下表 2-4 给出一个基于 SPARQL 查询语句单个实体查询的例子，其查询语义为已知书与 CRC 出版社的关系下查询所有 CRC 出版社出版书的标题和作者。

表 2-4 SPARQL 查询语句样例

PREFIX ex: <http://example.org/> SELECT ?title ?author WHERE {?book ex:publishedBy <http://crc-press.com/uri> . ?book ex:title ?title . ?book ex:author ?author }

该查询包括了三个部分，分别为关键字 PREFIX、SELECT 和 WHERE。关键字 PREFIX 声明一个命名空间，类似于 Turtle 语法中的@prefix，但是在 SPARQL 查询语言中，结束声明空间不需要句号。关键字 SELECT 之后列出的名字代表需要获取返回值的变量标识符，Select 行中明确提到的变量最终会显示在结果当中，没有提及的变量不会出现在结果集，在表 2-4 中变量为?title 和?author，即查询返回的是变量?title 和?author

对应的所有值。查询的实体以关键字 WHERE 开头，接着是一个以花括号包围的简单图模式。在表 2-4 的例子中包含了三个三元组，其表示方法类似于 Turtle，与 Turtle 表示法不同的是它还可以包含如?title 的变量标识符并且不需要所有的三元组都以句号结束，例如表 2-4 的例子中，最后一个三元组加不加句号都不影响查询的意义。

若以图数据库存储知识图谱，那么需要使用对应图数据库的查询方式查询知识图谱，如 Neo4j 使用 Cypher 查询语言查询图，ArangoDB 使用 AQL 查询语言查询图[29]。Neo4j 和 Cypher 将会在下节进行介绍。

2.2 大数据存储处理的相关技术

虽然有上述以基于表结构的关系型数据库存储知识图谱的方法，但是针对关系型数据库存储模式固定且知识图谱基于图数据结构的特点，目前流行使用非关系型数据库以基于图的结构存储知识图谱。接下来将会介绍目前主流的大数据存储处理的相关技术。

2.2.1 文档型数据库 MongoDB

MongoDB 是一款开源的、功能强大的、易于扩展、面向文档的 OLTP(On-Line Transaction Processing，联机事务处理过程)数据库。MongoDB 支持许多功能，如二级索引，全文索引和 MapReduce 操作等等。MongoDB 有三大特性：分别是灵活的文档模型，高可用的复制集和可扩展分片集群。灵活的文档模型是指 MongoDB 的存储模式不再固定，不再有关系型数据库中“行”的概念。MongoDB 基本存储单元是文档，文档还可以嵌套文档和数组。通过文档的形式 MongoDB 可以仅使用一条记录表示复杂的关系。文档由多个 key-value 键值对构成，key 一般为字符串，不能重复且不能含有空字符串、.、$和_，value 的数据类型则没有限制，还可以为文档。下图 2-3 给出一个文档模型的例子：通过在用户数据库中的 Orders 键值对查询 Orders 数据库中相关联的文档。

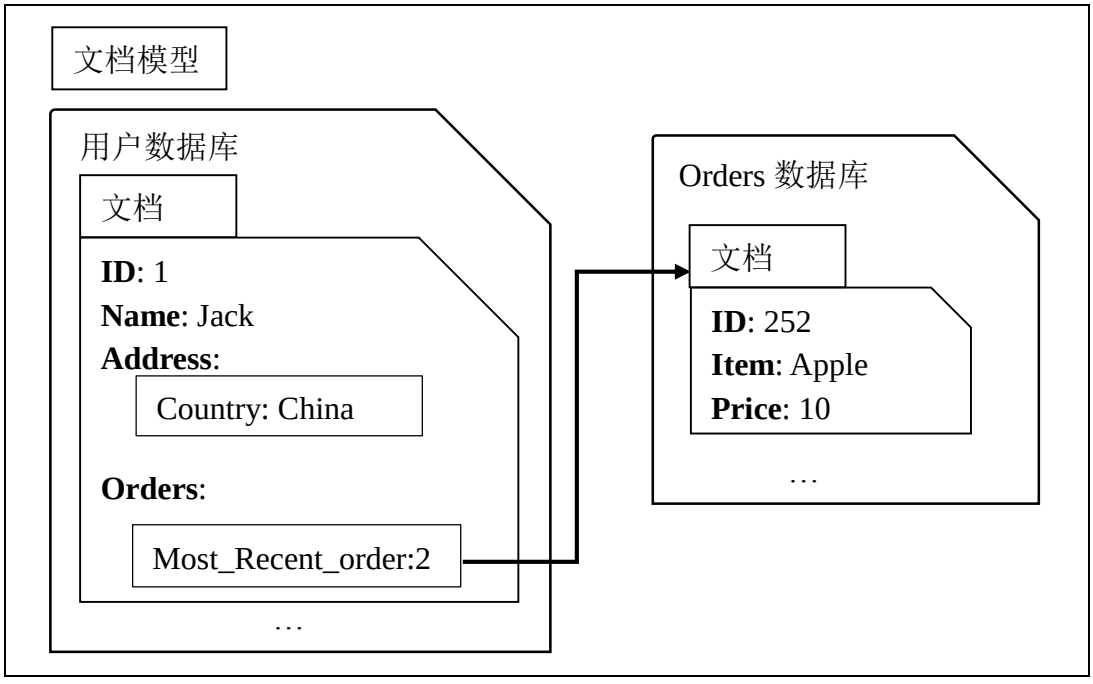


图 2-3 文档模型

图 2-3 的例子中，MongoDB 一共有 2 个数据库，一个为用户数据库，另一个为 Orders 数据库。用户数据库中的一个文档包含了 ID，Name，Address，Orders 等键值对，其中 Orders 这个键值对又与 Orders 数据库中的一个以 ID 为 252 的文档有关联。从图 2-3 可以看出文档模型接近真实的对象模型，对开发人员友好，模式不固定适应于灵活多变的需求。

高可用的复制集是指MongoDB采用一种保证数据高可靠和服务高可用的同步方法。MongoDB 通过基于 Raft 算法[30]选举出主节点，写操作时 MongoDB 先写到主节点然后异步同步到多个备用节点，各个主节点和备用节点之间一直保持心跳同步检测，当主节点故障时，MongoDB 马上在备用节点中自动选出新的主节点，读操作则可以在主节点和备用节点任意一个上读，异步复制过程十分高效。具体在写操作时，MongoDB 首先会让主节点将操作记录写入到本地服务器的日志中，然后从节点再读取本地服务器中的日志并存储在自己的数据库中，从而避免数据的丢失。但是一旦主节点出现故障且从节点没有完成主节点日志的读取和保存，即主节点和从节点的信息不一致，那么那些未读取的数据就会消失。下图 2-4 表示的是上述 MongoDB 采用复制集方法的同步流程。

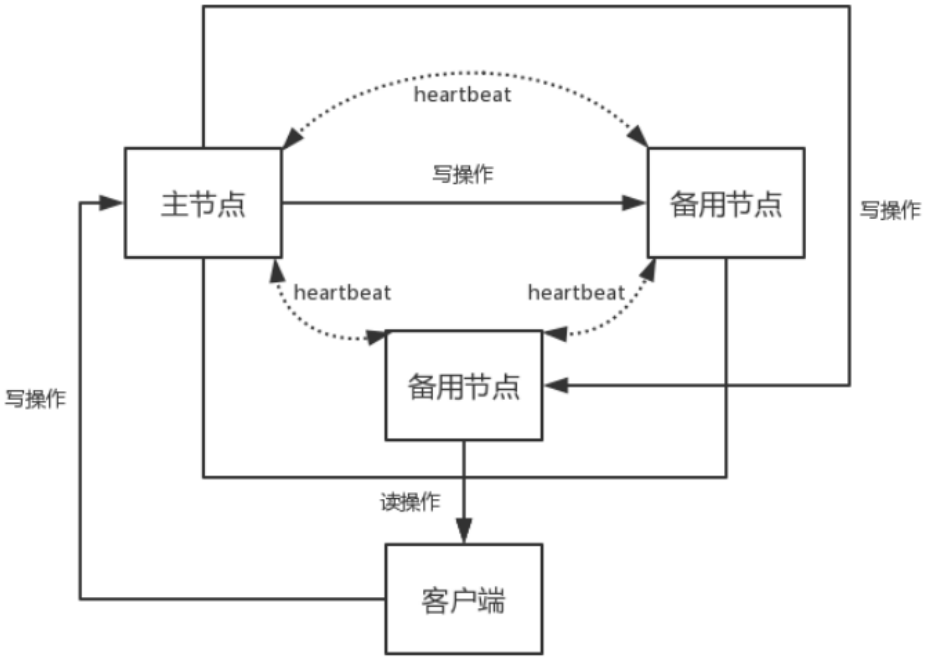


图 2-4 MongoDB 复制集方法同步流程

可扩展分片是指 MongoDB 通过可扩展分片的方式解决存储容量受限于单个主节点和写服务能力受限于单个主节点的问题。MongoDB 通过 Mongos 自动建立一个水平扩展的数据库集群系统，将数据库分表存储在各个分片节点上，同时每一个分片节点又都是一个复制集，从而保证数据的安全性。

MongoDB 与其他 NoSQL 相比存在以下的不足[31]：（1）一致性问题，分布式情况下会造成部分数据的丢失。（2）即使是在 MongoDB 宣传的适用场景下，其性能不高。（3）MongoDB 占用空间过大。（3）无法自动进行关联表的查询，不适用于关系多的查询。

2.2.2 图数据库 Neo4j

2.1.1 节提到，知识图谱另外的一种存储方式是使用图数据库存储。节点、边和属性是图数据库的核心概念。节点用于表示实体、事件等对象。边是指图中连接节点的有向线条，用于表示不同节点的关系。属性用于描述节点或者边的特性。基于此当使用图数据库存储知识图谱时可以使图数据库的节点表示知识图谱的实体，边表示知识图谱实体与实体之间的关系，属性表示实体的属性-属性值对。Neo4j 是当下稳定、高性能的非关系型图数据库，它具备完全的数据库 ACID（Atomicity，原子性；Consistency，一致性；Isolation，隔离性；Durability，持久性） 事务特性,保证数据的一致性。

Neo4j 有如下核心概念：（1）Node（节点）。节点是 Neo4j 中的基本元素，对比于关系型数据库，节点就是一行，节点经常被用来表示实体。（2）Relationships（关系）。一个关系连接 2 个节点，一个开始节点和一个结束节点。关系拥有进和出两种指向。关系的功能是组织和连接节点。（3）Properties（属性）。属性由键值对组成，键名是字符串，值的类型基本与 Java 的数据类型一致，还可以为数组。属性对比于关系型数据库就是字段，只有节点和关系才可以拥有 0 到多个属性。（4）Labels（标签）。标签一般是给节点加上一种类型，类似于分类，一个节点可以有多个类型，Neo4j 通过标签加速查询。同时标签也会应用在给属性建立索引或者约束的时候。标签的名称必须是非空的 unicode 字符串，标签的最大标记容量是 int 的最大值。（5）Paths（路径）。路径至少由一个节点，通过各种关系连接组成，作为一个图的查询或者遍历的结果（6）Traversal（遍历）。遍历是指查询时对查询条件以遍历的方式找到路径。在遍历时通过一个开始节点，然后根据 Neo4j 专门的 Cypher 查询语句遍历相关路径上的节点和关系，从而得到最终结果。（7）Indexes（索引）。遍历图是一个大量的随机读写操作，如果没有索引，每次遍历都是全图扫描，效率低下。通过在属性上建立索引可以大幅度提高查询性能，减少查询时间。（8）Constraints（约束）。约束可以定义在某个字段上，限制字段值唯一，创建约束时会自动创建索引。综上所述，对知识图谱使用 Neo4j 存储，只需将实体存储为节点，属性存储为 Properties，关系存储为 Relationships 即可。Neo4j 使用自己的查询语言 Cypher[32]进行图查询。下图 2-5 给出一个查询是否存在“张三-学生-李四”关系的 Cypher 查询语句的例子，。

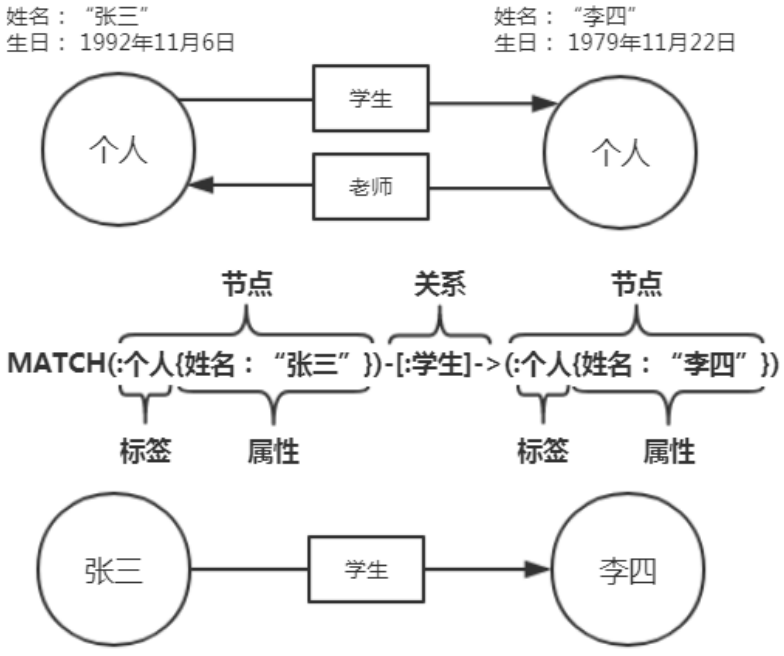


图 2-5 Neo4j 举例

图 2-5 定义了“张三”与“李四”之间的关系以及“张三”节点与“李四”节点的属性。然后使用 Cypher 查询语言的 match 语句匹配是否存在“张三-学生-李四”的关系，最后返回查询结果。Neo4j 适合有多个连接操作的复杂查询和不同节点之间的路径查询，但是对于全图搜索，简单查询，聚合查询，Neo4j 与其他数据库相比并不占优甚至性能较低，同时 Cypher 查询语言相比其他的查询语言会消耗大量的学习成本。目前开源的 Neo4j 版本为单机版，针对集群版本的 Neo4j 目前并未开源，若要使用集群版本需要支付大量的费用，经济成本较高。

2.2.3 分布式面向列的存储系统 HBase

HBase 是一个面向列的分布式存储系统，是 Google Bigtable 模型的开源实现，适用于在实时读写，并发随机访问超大规模数据集的应用场景。HBase 具有近似最优的写性能（能使 I/O 利用率达到饱和）和出色的读写性能，对于值为空的列不存储，充分利用磁盘空间，适合存放稀疏表。此外，HBase 还提供了过滤器功能来减少网络传输的数据量。相比于 Neo4j，虽然 HBase 只支持行原子性，但是它的“读-修改-写”操作能覆盖大部分使用场景并消除了在其他系统中经历过的死锁、等待问题。同时，HBase 的负载均衡和故障恢复对客户端是透明的，在生产系统中，系统的扩展完全自动化不需要人为的干预。下图 2-6 显示的是 HBase 系统架构图。

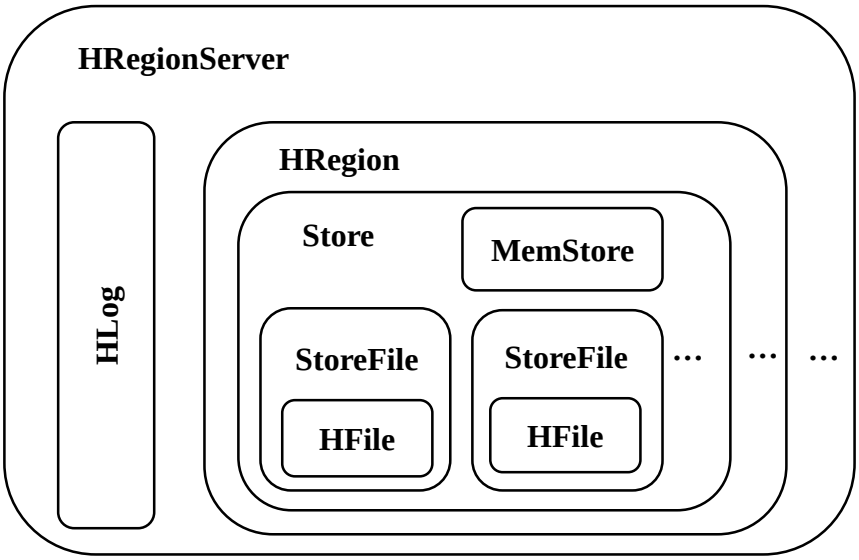
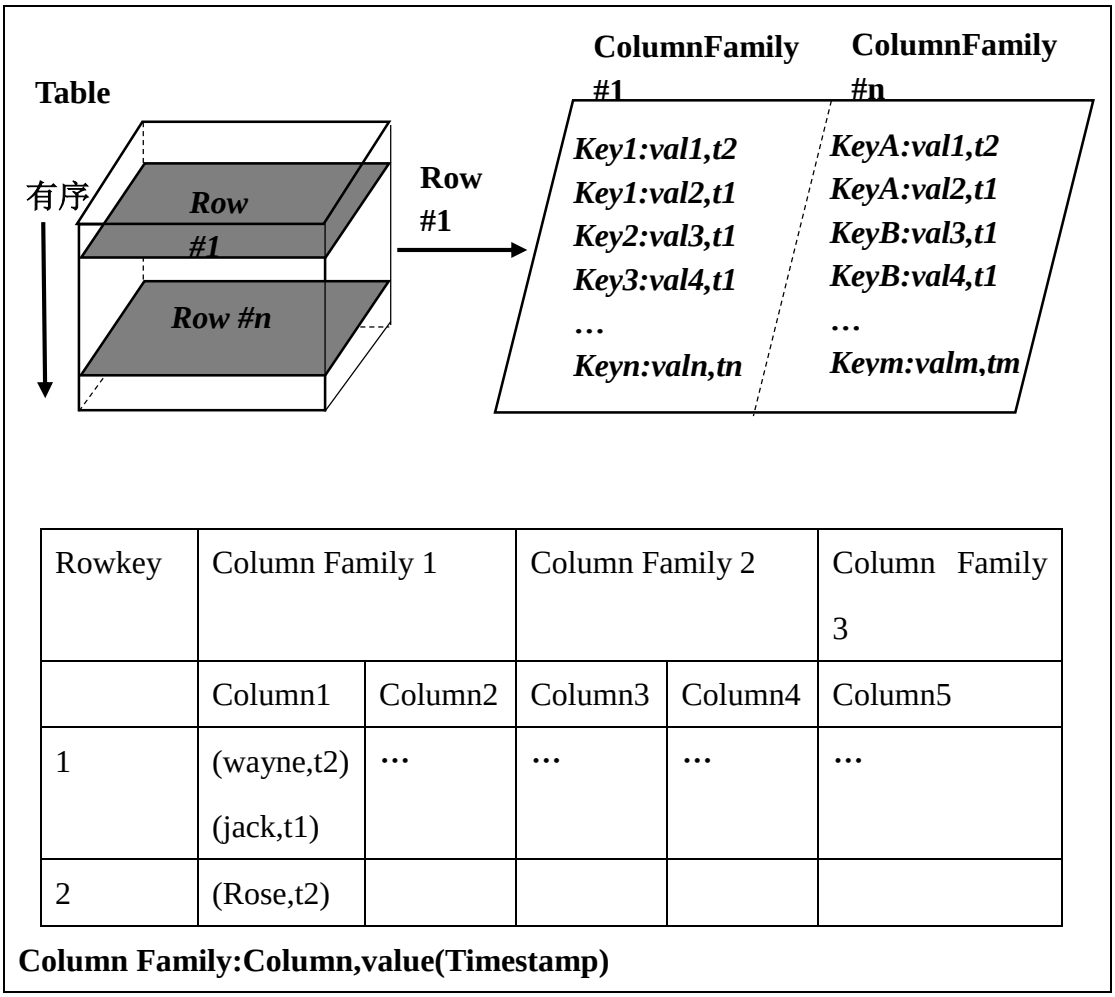


图 2-6 HBase 系统架构图

如上图 2-6，客户端主要与 HRegionServer（HBase 区域服务器）进行数据操作类通信。一个 HRegionServer 实际上就是一台拥有 HBase 服务的机器，HRegionServer 管理了一系列的 HRegion（HBase 区域）对象，每个 HRegion 对象对应表中的一个 Region（区域）。Region 是 HBase 数据管理的基本单位，其包括了 HBase 表内完整的一行。HRegion 由多个 Store（仓库）组成，每个 Store 对应表中的一个 Column Family（列簇）的内容。每个 Store 包含一个 MemStore（内存仓库）和 0 到多个 StoreFile（仓库文件），StoreFile 最后存储时对应为 HFile（HBase 文件），HFile 存储在 HDFS（Hadoop Distributed File System,Hadoop 分布式文件系统）的中。HLog（HBase 日志）是 HBase 中的日志文件，

数据在写入时，首先写入 HLog。MemStore 是一个有序的内存缓冲区，数据写入到 HLog 之后再写入到 MemStore 中，当 MemStore 满了之后 Flush（洗牌）成一个 StoreFile。

HBase 是以表形式存储数据，主要由 4 部分组成：分别是 Rowkey（行键），Column Family，Column（列），TimeStamp（时间戳）和 Cell（单元格）。行键是用来检索记录的主键，列簇是 HBase 表的模式一部分，访问控制与磁盘和内存的使用统计都是在列簇上进行。时间戳是 HBase 控制数据版本的方式，时间戳可以由用户自己设置或者由 HBase 自动赋值。单元格是由行键、列簇和列确定的存储单元，单元格中数据没有类型，全部都是以字节的方式存储。在 HBase 中记录是按照行键的字典序排列存储，一行中可以有多个列簇，一个列簇可以包括多个列，其中每列可能有多个版本（以时间戳区分）。下图 2-7 是之前描述的 HBase 数据模型。



Rowkey	Column Family 1		Column Family 2		Column Family 3
	Column1	Column2	Column3	Column4	Column5
1	(wayne,t2) (jack,t1)	…	…	…	…
2	(Rose,t2)				

Column Family:Column,value(Timestamp)

图 2-7 HBase 数据模型

从图 2-7 中给出的例子可以看出，对于行键，HBase 使用字典序的方式进行排序，对于单元格的内容，HBase 首先根据插入或者更新的时间戳倒序排序，然后再对 value 值根据字典序排序。HBase 适合存放大且稀疏的表，针对大且稀疏的表，使用 HBase 往往能带来良好的存储和查询体验。

HBase 的可扩展性相比 MongoDB 和 Neo4j 十分优秀，它自动将 Region 分配给集群的不同 RegionServer（区域服务器）实现扩展。当系统负载高时，HBase 可以通过简单的添加机器实现水平切分扩展，同时 HBase 与 HDFS 的无缝集成可以避免单点故障的

发生，保障数据的可靠性。因为 HBase 的日志存放在 HDFS 中，所以当某一节点发生故障，基于 HDFS 良好的容错性，HBase 可以从其他节点获取发生故障节点的数据。若 HBase 在数据节点上发生故障，HBase 只会造成短暂的宕机，虽然读性能在数据的恢复过程中会受到影响，但是数据的一致性完全可以得到保证。若 HBase 在主节点崩溃，HBase 集群仍然能以稳定状态模式管理读取与写入请求，只是无法执行需要主节点介入的如负载均衡调整的协调工作，此时只需要替补主节点快速介入，那么集群就能重新保持良好的运作状态。

HBase 的缺点是查询方式简单，不支持条件查询，只支持行键查询、行键的范围查询或全表扫描，并且自身不能建立二级索引。

2.2.4 分布式内存计算引擎 Spark

Apache Spark[33]是目前流行的针对大规模数据处理的快速通用计算引擎。相比较与 MapReduce 框架，Spark 可以将 Job（作业）中间的输出结果保存在内存中，不需要重新再读写磁盘，同时，Spark 使用了统一抽象的 RDD（Resilient Distributed Dataset,弹性分布式数据集）数据结构，使得 Spark 不仅比 MapReduce 拥有更多的方法，同时还可以以基本一致的方式应对不同的大数据处理场景[34]。

RDD 是 Spark 的核心数据结构。其实简单的说，RDD 就是一个简单的分布式元素集合,是创建后不可变的数据结构，支持跨集群操作。Spark 中所有的工作都描述为两种：1.创建新的 RDD 或转换已存在的 RDD。2.在 RDD 上调用一些操作来计算结果。RDD 的操作分为两种：1. Transformations（转换）。Transformations 仅能返回一个新的 RDD。2. Actions（动作）。Actions 可以返回结果值或者对结果进行存储。RDD 使用的是一种懒加载模式，即只有当执行 Actions 操作时才会启动计算执行 Transformations 操作，在 Transformations 操作时 Spark 不会去执行，只会记录这个操作，只有遇到 Actions 操作时才真正启动计算。下图 2-8 是 RDD 的计算流程图。

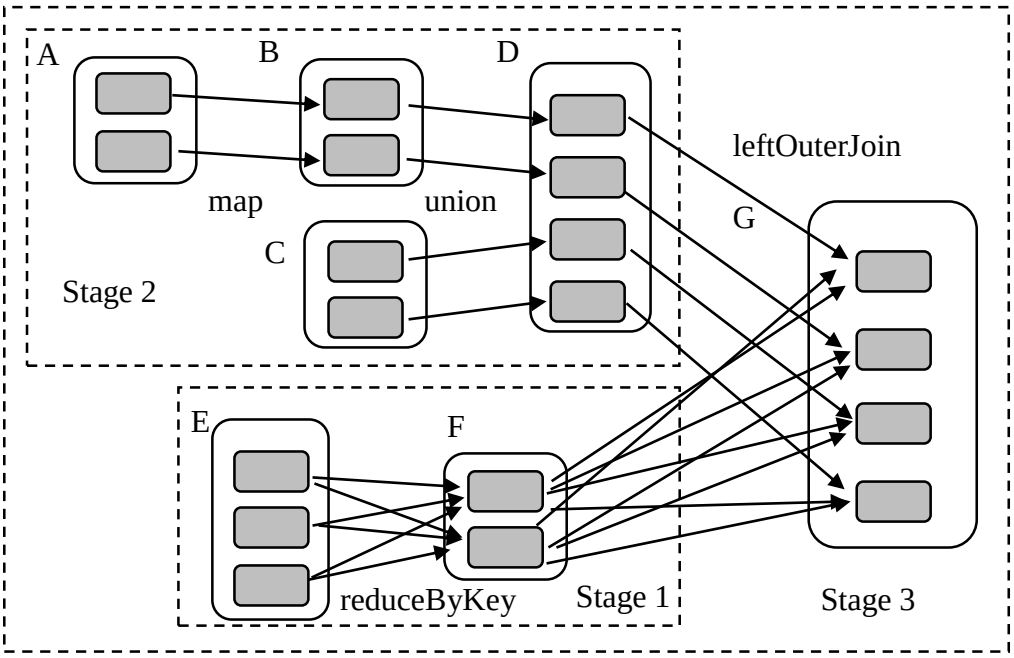


图 2-8 RDD 计算流程图

每个 Action 的计算会生成一个 Job，一个 Job 根据 shuffle 拆分成若干个 Stage（阶段）。如图 2-8，图中一个大圆角矩形表示一个 RDD，一个带阴影的小圆角矩形代表 RDD 的 partition（分区）。图中有 reduceByKey（key 聚集），leftOuterJoin（左连接）两个 shuffle 操作，因此一个 Job 被划分成三个 Stage。Stage 1 中的 RDD E 经过 reduceByKey 的 shuffle 操作转换成 RDD F。Stage 2 中 RDD A 先经过 map（映射）操作转换为 RDD B，然后 RDD B 和 RDD C 再做 union（并）操作转换为 RDD D。Stage 3 中 RDD D 和 RDD F 经过 leftOuterJoin 的 shuffle 操作转换成 RDD G。虽然 Spark 的性能优于 Hadoop，但是 Spark 并不会取代 Hadoop，Spark 与 Hadoop 实际上是一种互补的关系，Spark 可以无缝与 HDFS 进行连接操作。目前 Spark 常用于实时计算，数据分析等操作。

2.3 本章小结

本章主要分为两个部分介绍，第一部分主要以知识图谱的数据模型和知识图谱的查询语言介绍知识图谱存储查询的相关理论。第二部分主要介绍大数据存储处理的相关技术，如 MongoDB，Neo4j，HBase 和 Spark。论文的后几章将会对上述的技术进行知识图谱存储查询的性能比较，设计满足知识图谱存储查询性能扩展性的存储模式和分布并行查询引擎，最后实现基于大数据平台的知识图谱存储访问原型系统并进行性能比较。

第三章 知识图谱存储访问系统存储模式的设计

本章主要介绍本文基于 Big Table 模型开源实现的 HBase 设计满足知识图谱存储可扩展要求的分布聚集存储模式。通过对表预分区和对逐行存储的实体集合进行随机前缀的操作使得知识图谱存储具有总体存储负载分布均衡、局部存储聚集的特点。同时，随机前缀也能够对同类型的实体均匀地分布到节点存储，并在单个节点上按实体类别聚集。该存储模式不仅减少数据冗余还能保证系统数据负载均衡状态保证知识图谱的存储性能。

3.1 数据存储系统的选择

面对知识图谱存储查询可扩展的需求，本文选择基于 Big Table 模型的 HBase 作为知识图谱的数据管理系统。主要原因：1. 存储性能优异。面对大规模知识，基于 LSM[35] 树实现的 HBase 拥有更快的存储速度，相比于其他两种数据库更加利于大规模知识的存储。2. 可扩展性优异。HBase 通过简单的添加机器实现水平切分扩展，自动提升系统的存储和查询能力。面对大规模知识存储，单机节点的 Neo4j 无法支撑系统大数据量的存储要求，MongoDB 则拥有数据易丢失的缺点。3. 较优的读性能。面对大规模数据，HBase 的行键查询依旧可以达到毫秒级。虽然面对小规模数据，Neo4j 和 MongoDB 在知识图谱常使用的单个实体查询和连接查询下拥有比 HBase 更加优秀的查询性能，但是面对大规模数据，Neo4j 无法支持且 MongoDB 性能急剧下降，而 HBase 的查询性能依旧稳定。

为了充分说明 MongoDB，Neo4j 和 HBase 的存储查询水平扩展性的性能，本节对三种数据库进行存储性能和查询性能的比较。实验测试环境为：由 3 台 core i7 处理器，内存 8G 机器搭起来的实验室集群，MongoDB 使用的版本为 3.4.6，Neo4j 使用的版本为 3.1.5，HBase 使用的版本是 1.2.0-cdh5.13.0。

在进行存储性能的测试时，MongoDB 以 key-value 的形式将 RDF 三元组存储在文档中。HBase 表中的一行表示一个 RDF 三元组，每行有三列分别存储 RDF 三元组的主语，谓语和宾语。Neo4j 则将 RDF 三元组中的“实体-关系-实体”和“实体-属性-属性值”以图形式通过 load csv[20]方式存储。本文的数据均来自标准数据集 LUBM[36]。本节将会以十万个 RDF 三元组，一百万个 RDF 三元组和一千万个 RDF 三元组进行存储时间的比较，比较结果如下表 3-1 所示，表 3-1 中 N/A 表示数据存储时间无法估计。

表 3-1 三种数据库存储性能比较

	十万 RDF 数据集	一百万 RDF 数据集	一千万 RDF 数据集
MongoDB	5112 ms	61759 ms	455550 ms
Neo4j	5824250 ms	N/A	N/A
HBase	4928 ms	59378 ms	394269 ms

从表 3-1 可以看到，随着数据量的增加，三种数据库的存储时间都有成倍的增加。但是在现有机器配置下，当 RDF 三元组个数达到一百万时，Neo4j 的存储时间无法估计并且在存储十万个 RDF 三元组时 Neo4j 的存储时间也远超出相同数据量级下的 MongoDB 和 HBase 的存储时间。基于 LSM 架构实现的 HBase 理论上写入速度相比与 Mongo

DB 要快，实验数据也证明在各个数据量级下 HBase 的写入性能优于 MongoDB。实验验证，面对大数据量的 RDF 三元组，HBase 的存储性能优于 MongoDB 与 Neo4j。

在进行查询性能的测试时，本节在不同数据量级的 RDF 三元组下对三种数据库进行简单查询和连接查询的查询性能比较。简单查询是指单个条件的查询，如以主语条件、谓语条件和宾语条件进行查询。连接查询已在 2.1.2 节中进行说明，不节不再赘述，本节的连接查询包括两个实体和三个实体间的查询。下表 3-2，3-3，3-4 是分别在各个数据量级的 RDF 三元组下的简单查询和连接查询的实验。实验得出的查询时间为在相同查询类型下 10 个不同查询语句的平均查询时间。

表 3-2 十万 RDF 数据集的查询时间

	简单查询			连接查询	
	主语条件	谓语条件	宾语条件	两个实体	三个实体
MongoDB	172 ms	4634 ms	593 ms	325 ms	484 ms
Neo4j	255 ms	1488 ms	92 ms	268 ms	322 ms
HBase	555 ms	1011 ms	818 ms	749 ms	823 ms

表 3-3 一百万 RDF 数据集的查询时间

	简单查询			连接查询	
	主语条件	谓语条件	宾语条件	两个实体	三个实体
MongoDB	526 ms	36048 ms	4177 ms	2706 ms	3816 ms
Neo4j	N/A	N/A	N/A	N/A	N/A
HBase	638 ms	1449 ms	1116 ms	1039 ms	1427 ms

表 3-4 一千万 RDF 数据集的查询时间

	简单查询			连接查询	
	主语条件	谓语条件	宾语条件	两个实体	三个实体
MongoDB	5060 ms	320644 ms	37401 ms	26084 ms	36610 ms
Neo4j	N/A	N/A	N/A	N/A	N/A
HBase	708 ms	8604 ms	4801 ms	4942 ms	8657 ms

通过上述三种情况的查询时间比较可以明显发现随着数据量的增加，简单查询和连接查询的查询时间均有相应的增加。从表 3-2 可以明显看出在十万 RDF 三元组条件下，无论是简单查询和连接查询，Neo4j 的查询时间基本都优于 MongoDB 和 HBase，Neo4j 通过自己优秀的查询图算法，能快速地在图中查询并及时返回结果。由于在一百万和一千万的数据集下 Neo4j 的存储未能完成，因此在这两个数据集下无法对 Neo4j 进行查询时间测试。依据 Nayak 等[37]的发现：随着节点和关系的增加，Neo4j 的性能会急剧下降，当节点和关系达到千万级别时查询效率低下不能接受，可见即使在分布式情况下，面对大数据量的知识图谱时，Neo4j 的查询性能也并不优异。从三张表的结果来看，随着数据量的增加，HBase 的查询性能优于 MongoDB，尤其是基于主语条件的简单查询，HBase 充分利用其行键查询的优势，在实验测试的数据量级下查询时间均小于 1 秒。对于连接查询，由于 MongoDB 和 HBase 本身都不支持连接操作，因此本节使用嵌套连接的方式在 MongoDB 和 HBase 中进行连接查询。通过实验数据发现虽然 HBase 的连接查询时间

短于 MongoDB 的连接查询时间，但是其查询性能依旧不理想，对于实时查询而言需要改进。

综合上述三种数据库存储和查询知识图谱的性能，实验验证面对知识图谱存储查询可扩展的需求，基于 Big Table 模型的 HBase 适合作为知识图谱的数据管理系统。上述三种数据库在各个方面都有其优点和缺点。在存储时间方面，由于 HBase 架构的优越性，面对海量数据，HBase 相比 MongoDB 和 Neo4j 都有不小的存储速度优势。在查询时间方面，Neo4j 在查询多个关系时速度优势明显，但是随着数据量级的增加，Neo4j 无法有效存储大数据量级的数据，MongoDB 的查询性能也有明显下降，而 HBase 无论是在简单查询还是连接查询下其查询性能均优于 MongoDB 和 Neo4j 且若对 HBase 的数据模型进行知识图谱存储模式的改进和设计并行查询方案，面对大规模知识图谱，HBase 在知识图谱的存储和查询性能上会有极大的提升空间。

3.2 分布聚集存储模式的设计

面对实体和关系数目庞大且不断增加的知识图谱，知识图谱存储访问系统需要设计一个可扩展的存储模式提升系统的存储性能。目前基于 HBase 的以 RDF 三元组结构存储知识图谱的研究有很多。Li 等[17]提出对每个 RDF 三元组的谓语建立两张表存储该 RDF 三元组的方案。如下图 3-1，对于 RDF 三元组集合“Lecture-teacherOf-Course”，以谓语“teacherOf”建立两张表（SO，OS）。

SO

Row Key	ColumnFamily:cf
Lecture0	Cf:Course0 Cf:Course1
Lecture1	Cf:Course1 Cf:Course3

OS

Row Key	ColumnFamily:cf
Course0	Cf:Lecture0 Cf: Lecture1
Course1	Cf: Lecture1 Cf: Lecture3

图 3-1 基于谓语“teacherOf”的存储方案

在上图 3-1 中，SO 表的 Row key 以 RDF 三元组的主语“Lecture”表示，SO 表中 Cell 的值以 RDF 三元组的宾语“Course”表示，对于 OS 表，RDF 三元组的主语“Lecture”为 Cell 的值，RDF 三元组的宾语“Course”为 Row key。该存储模式旨在借助 HBase 高性能行键查询的特性以两张表的形式分别在行键中存储三元组的主语和三元组的谓语，加快查询速度。但是该存储模式存在重复存储，浪费大量的存储空间的问题，同时由于 HBase 以字典序的顺序对行键进行排序，该存储模式也易造成数据热点，影响存储和查询性能，除此之外，若出现谓语个数为上万个的情况下，那么表会达到上万张，不利于管理。Franke 等[6]针对 Li 等[17]设计的基于谓语的 RDF 三元组存储进行改进,提出了使用多列存储 RDF 三元组的存储模式方案，如下图 3-2。Franke 等[16]总共使用两张表（SPO，OPS）存储所有 RDF 三元组集合，每张表只有一个列簇 cf，列簇的列对应的是谓语。在 SPO 表中 RDF 三元组的主语为 Row key，RDF 三元组的宾语为 Cell 的值。在 OPS 表中 RDF 三元组的宾语为 Row key，RDF 三元组的主语为 Cell 值，该方案相比于 Li 等[17]设计的方案在保证查询效率的同时还减少了表的生成，便于数据的管理。但是该方案依旧会造成数据的大量冗余和数据热点问题。

SPO

Row key	**cf:type**	**cf:name**	**cf:memberOf**	**…**
<C>	{<Student>}	{“Craig”}	{<IEEE>}	…
<S>	{<Student>}	{“Sam”}	{<ACM>}	…

OPS

Row key	**cf:type**	**cf:name**	**cf:memberOf**	**…**
<Student>	{<C>,<S>}			…
“Craig”		{<C>}		…
“Sam”		{<S>}		…
<IEEE>			{<C>}	…
<ACM>			{<S>}	…

图 3-2 多列存储 RDF 三元组的存储方案

本文借鉴 Franke 存储模型的思路并充分利用 HBase 稀疏、分布式、一致、多维排序的特性，设计出单表多列簇的分布聚集存储模式。该存储模式不仅能充分避免数据的冗余，还具有总体存储负载分布均衡、局部节点聚集存储特点，满足知识图谱存储可扩展性的需求。该存储模式具体如下图 3-3 所示。

Row key	**Attributes**		**Objects**	
	属性 1	**…**	**关系 1**	**…**
0000:类型:实体 1	属性值 1	…	0999:类型:实体 6	…
…	…	…	…	…
0999:类型:实体 6	属性值 2	…	…	…
1000:类型:实体 x **…** **8999:类型:实体 n**	…	…	…	…
9000:类型:实体 2	属性值 3	…	9999:类型:实体 8	…
…	…	…	…	…
9999:类型:实体 8	属性值 4	…	…	…

图 3-3 本文设计的分布聚集存储模式

如上图所示，本文设计的分布聚集存储模式只使用一张大表存储所有的 RDF 三元组集合，其中每一行存储某个实体拥有的所有属性值和关系对象，即存储一个完整的实体对象。Row key 存储主语，即实体，以谓语表示的属性和关系分别存储在列簇“Attributes”和“Objects”列中，Cell 中的值存储对应的属性值或实体，其中实体表示与行键的实体表示一致。将属性与关系用两个列簇存储不仅便于管理，而且可以减少查询时内存加载的列数量，利于将更多的实体加载到内存，加快查询速度。为了满足知识图谱存储的要求，该存储模式的 Row key 格式设计如下：“random:类型:实体”。“random”为四位随机数前缀，“:”为本文自定义的分隔符，“类型”表示实体所属的概念，“实体”对应 RDF 三元组的主语。由于 HBase 以字典序排序，当遇到大量对相同首字母的实体查询时，该查询可能会对集群某一两个节点产生大量访问，造成数据热点问题，严重影响存储性能。

为了避免数据热点，本文在建表时先对表进行预分区操作，预分区操作是旨在集群中较为均匀的预建好一部分具有 startkey（起始行键）和 endkey（终止行键）的 Region。预分区操作不仅可以避免 Region split 带来的资源消耗，而且可以并行存储数据，提高数据的存储效率。对于 HBase 集群，一台机器可以存储多个 Region，而一个 Region 只能存于一台机器。虽然 Region 在存储时随机指定节点，但是 HBase 集群会尽量让 Region 均匀地分布在每个节点。本文对表预分成 10 个 Region，每个 Region 大小为 10GB 且 Region 之间以 1000 相隔，即每个 Region 的起始-终止行键为：0000-0999，1000-1999，2000-2999，3000-3999，4000-4999，5000-5999，6000-6999，7000-7999，8000-8999，9000-9999。下图 3-4 为本文对 HBase 的预分区示意图。

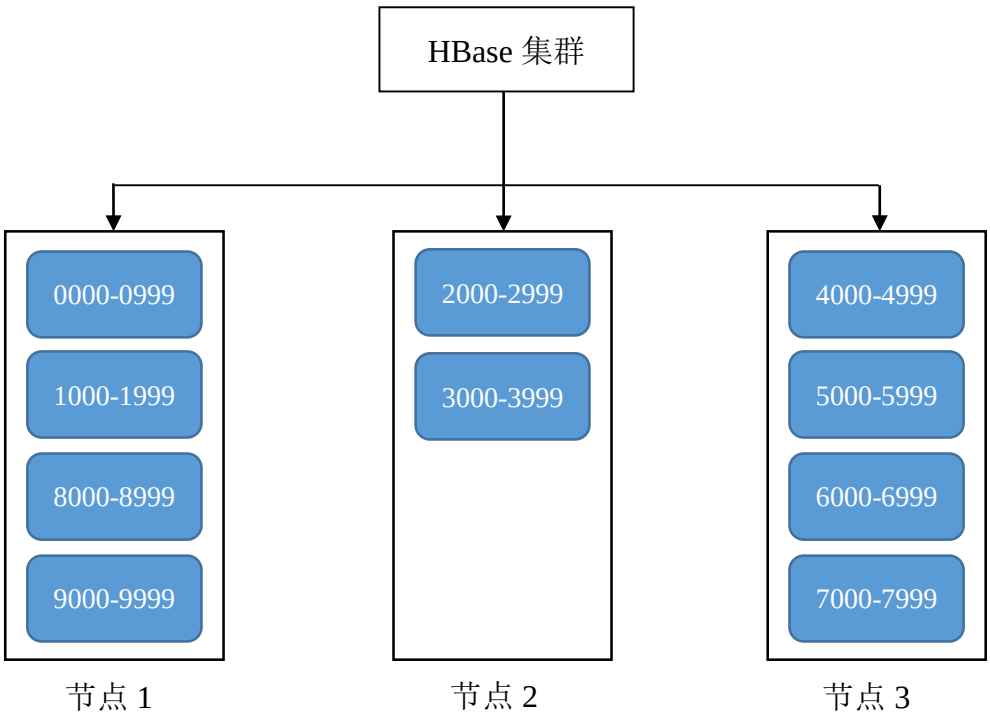


图 3-4 HBase 预分区存储示意图

Region 个数的设定可以根据系统的需求设置。一般计算每台机器最多存储 Region 数量的公式如下：其中 total memory size 指的是机器的内存大小，total memstore fraction 指的是 HBase 分配给 memstore 的比例，memstore 指的是 HBase 的存储缓存，memstore size 指的是 memstore 的大小，number of column family 指的是列簇的个数。

$$(\text{total memory size}) * \frac{(\text{total memstore fraction})}{(\text{memstore size})} * (\text{number of column family}) \quad (1)$$

为了保证 RDF 三元组存储时能均等的命中预分区时建立的 Region，本文还使用四位随机数作为 Row key 的前缀。行键起始设置为随机前缀可以打乱实体根据字典序存储的存储顺序，实体最终存储到其随机前缀对应的 Region 中。面对海量的数据，基于随机前缀设计的行键不仅可以总体上将所有的实体在集群中分布存储，还尽量能将同类型的实体均匀地分布到各个 Region 中存储，使每个 Region 按实体的类型聚集。按实体的类型聚集的示意图如下图 3-5。假设 RDF 三元组集合只有 4 种类型，那么每个 Region 都会尽量以相同的分布存储这 4 种类型的实体。

图 3-5 实体类型聚集示意图

除此之外，基于设计的分布聚集存储模型，为了保证 Region 分割后知识图谱存储访问的性能，本文还实现了一种 Region 的分割策略。当 Region 大小达到上限时，该策略会以 Region 行键个数的中位数进行分割，原 Region 的起始行键-原 Region 的中位数行键（不包含）作为新的 Region1 的起始-终止行键，原 Region 的中位数行键-原 Region 的终止行键作为新的 Region2 的起始-终止行键。新的 Region1 和新的 Region2 可以无缝地仍以分布聚集存储模式存储数据。

明显看出，本文设计的分布聚集存储模式不仅减少数据重复存储造成的数据冗余，而且避免多张表数据插入、更新操作时造成的数据不一致，还可以充分利用集群的性能，提高并行性并保证负载均衡状态，使实体存储具有总体存储负载分布均衡、局部节点聚集存储的特点，充分满足知识图谱存储的要求。下表 3-5，3-6，3-7，3-8 是本文设计的分布聚集存储模式与 Franke 等[6]设计的存储模式在不同数量级下存储和查询的性能比较。

表 3-5 Franke 存储模型与本文的分布聚集存储模型存储时间的比较

	十万 RDF 数据集	**一百万 RDF 数据集**	**一千万 RDF 数据集**
Franke 存储模型	14362718 ms	132287049 ms	1719414739 ms
分布聚集存储模型	69400 ms	639836 ms	6928636 ms

表 3-6 十万 RDF 数据集下 Franke 模型与本文模型查询时间的比较

	简单查询			**连接查询**	
	主语条件	**谓语条件**	**宾语条件**	**两个实体**	**三个实体**
Franke 存储模型	494 ms	680 ms	544 ms	510 ms	614 ms
分布聚集存储模型	505 ms	669 ms	808 ms	820 ms	900 ms

表 3-7 一百万 RDF 数据集下 Franke 模型与本文模型查询时间的比较

	简单查询			**连接查询**	
	主语条件	**谓语条件**	**宾语条件**	**两个实体**	**三个实体**
Franke 存储模型	753 ms	1477 ms	703 ms	850 ms	1904 ms
分布聚集存储模型	830 ms	1469 ms	1206 ms	1339 ms	2427 ms

表 3-8 一千万 RDF 数据集下 Franke 模型与本文模型查询时间的比较

	简单查询			**连接查询**	
	主语条件	**谓语条件**	**宾语条件**	**两个实体**	**三个实体**
Franke 存储模型	1050 ms	5753 ms	1014 ms	1813 ms	4548 ms
分布聚集存储模型	1708 ms	5802 ms	4305 ms	5142 ms	8757 ms

表 3-5 显示 Franke 存储模型的存储时间远长于本文提出的分布聚集存储模型，其主要原因在于 Franke 存储模型的 OPS 表本质是一个建立二级索引的过程，在存储实体到

OPS 表时不仅需要遍历已建立的 SPO 表，还需要判断该表中是否已存储对应 RDF 三元组主语的情况，所以 OPS 表的建表时间会明显长于 SPO 表的建表时间。Franke 存储模型的 SPO 表的建表时间根据表的数据量级依次为 6393ms，56754ms，858642ms，该结果验证 OPS 表的构建是造成 Franke 存储模型的存储时间长于本文设计的分布聚集存储模型的主要原因。表 3-6，3-7 和 3-8 显示的是两种存储模型在不同数据量级下的查询时间。由于 Franke 存储模型构建倒排索引 OP 表，而本文提出的分布聚集存储模型除了主键的自身索引外并没有构建其他索引，因此对于 RDF 三元组的宾语和谓语的查询只能使用 HBase 自身提供的过滤器进行过滤，而过滤器过滤是一个扫描全表的操作，查询效率低下。实验数据也验证，本文提出的分布聚集存储模型在基于宾语和谓语的单个条件查询下的简单查询和连接实体查询其查询时间长于 Franke 存储模型。所以就查询效率而言，Franke 存储模型优于本文提出的分布聚集存储模型。

本文提出的分布聚集存储模型虽然在查询速度上慢于 Franke 存储模型，但是其在存储效率，系统并行性和数据负载均衡方面优势十分明显，满足知识图谱存储可扩展的要求。该模型的查询效率不及 Franke 存储模型的主要原因在于没有充分利用集群可以并行查询的特性，因此为了弥补分布聚集存储模式查询效率的不足，本文基于分布聚集存储模式提出了两种并行查询方案，这两种查询方案将在下一章节进行详细的讲解。

3.3 本章小结

基于 Big Table 模型开源实现的 HBase 的优秀存储、查询和扩展性性能，本文利用 HBase 稀疏、分布式、一致、多维排序的特性，设计出具有单表多列簇的分布聚集存储模式。通过对表预分区和对逐行存储的实体集合进行随机前缀的操作使得知识图谱存储具有总体存储负载分布均衡、局部存储聚集的特点。同时，随机前缀也能够对同类型的实体均匀地分布到节点存储，并在单个节点上按实体类别聚集。该存储模式相比常用的存储模式能大幅地提高存储效率，提升系统的并行度并保证系统的负载均衡状态，满足知识图谱存储可扩展的要求，但是在查询性能上还有所欠缺，因此下一章节本文会基于分布聚集存储模式提出两种并行查询的查询引擎。

第四章 知识图谱存储访问系统分布并行查询引擎的设计

面对规模庞大的知识图谱，为了满足知识图谱又快又准的查询需求，知识图谱需要设计分布并行查询方案以并行扫描查询树的方式加速查询，保证知识图谱的查询性能。本章主要介绍本文在 Group-By 模式下设计两种知识图谱分布并行查询引擎。基于上章提出的分布聚集存储模式，本章节设计出采用分布式内存迭代技术的查询引擎 MIQE（Memory Iteration Query Engine）和采用倒排索引技术的查询引擎 IIQE（Inverted Index Query Engine）。MIQE 以 Spark 计算模型为基础并行地在集群内存中以过滤和连接操作迭代查询以抽象集合表示的实体。IIQE 是以倒排索引的方式在集群中并行查询对应查询条件的实体。上述两种分布并行查询引擎都旨在通过并行方式提升知识图谱系统的读性能，加快知识图谱查询速度。

4.1 采用分布式内存迭代技术的查询引擎 MIQE

本节采用分布式内存迭代技术以 Spark 计算模型为基础设计查询引擎 MIQE。对于复杂的查询若只使用 HBase 的过滤器方式进行查询，则其会以全表顺序扫描的方式对相应的查询条件进行查询，面对海量数据，过滤器方式的查询性能低下，严重影响用户体验。虽然 HBase 支持使用 MapReduce 计算框架并行查询数据，但 MapReduce 会将中间结果存放在磁盘，从而产生大量的磁盘 I/O 操作，消耗系统运行时间，并不适合实时查询的应用场景。若将数据存储到内存则可避免磁盘的操作，因此本节想到使用 Spark 计算模型并行查询知识图谱。Spark 是目前流行的一种通用分布式大数据计算平台，其通过细粒度的任务调度和对内存的充分利用能很好地解决大数据领域数据的计算问题，由于 Spark 底层支持 HDFS（Hadoop Distributed File System,Hadoop 分布式文件系统）的文件格式，所以 Spark 可以很好的与 HBase 进行结合。Spark 与 MapReduce 有如下几处不同：1. Spark 的所有运算过程都基于内存，因此 Spark 可以避免大量磁盘的 I/O 操作。2. Spark 类似于 MapReduce 模型，提供了 map 阶段和 reduce 阶段，但是在两个阶段 Spark 提供了更丰富的算子，因此理论上 Spark 的计算速度比 MapReduce 快许多。如上所述 Spark 的特性，采用分布式内存迭代技术以 Spark 计算模型为基础设计的查询引擎 MIQE 不仅可以并行查询还能避免磁盘 I/O 的消耗，满足知识图谱查询的性能需求。

本节设计的查询引擎 MIQE 可以无缝地支持上章提出的以分布聚集存储模式存储知识图谱的存储系统，具体地查询引擎设计方案如下图 4-1 所示。分布并行查询系统由数据处理层和数据存储层构成。存储时，知识图谱以设计的分布聚集存储模式存储到数据存储层中的 HBase，查询时，知识图谱先加载到数据处理层中的 Spark 内存，然后 Spark 根据查询条件操作算子查询并返回结果。基于分布聚集存储模式，MIQE 可以拥有良好的并行查询性能，其主要原因在于：Spark 加载 HBase 数据时以 HBase 的 Region 作为单元，即 Region 的个数表示 Spark 分区的个数，同时又由于 Region 内部存储的实体类型拥有局部聚集的特性，所以在有集群本身拥有良好的性能的前提下，MIQE 设计方案不仅可以避免数据倾斜导致的 shuffle[33]时间过长，还可以同时并行对所有 Region 查找符合查询条件的实体，提升查询性能。

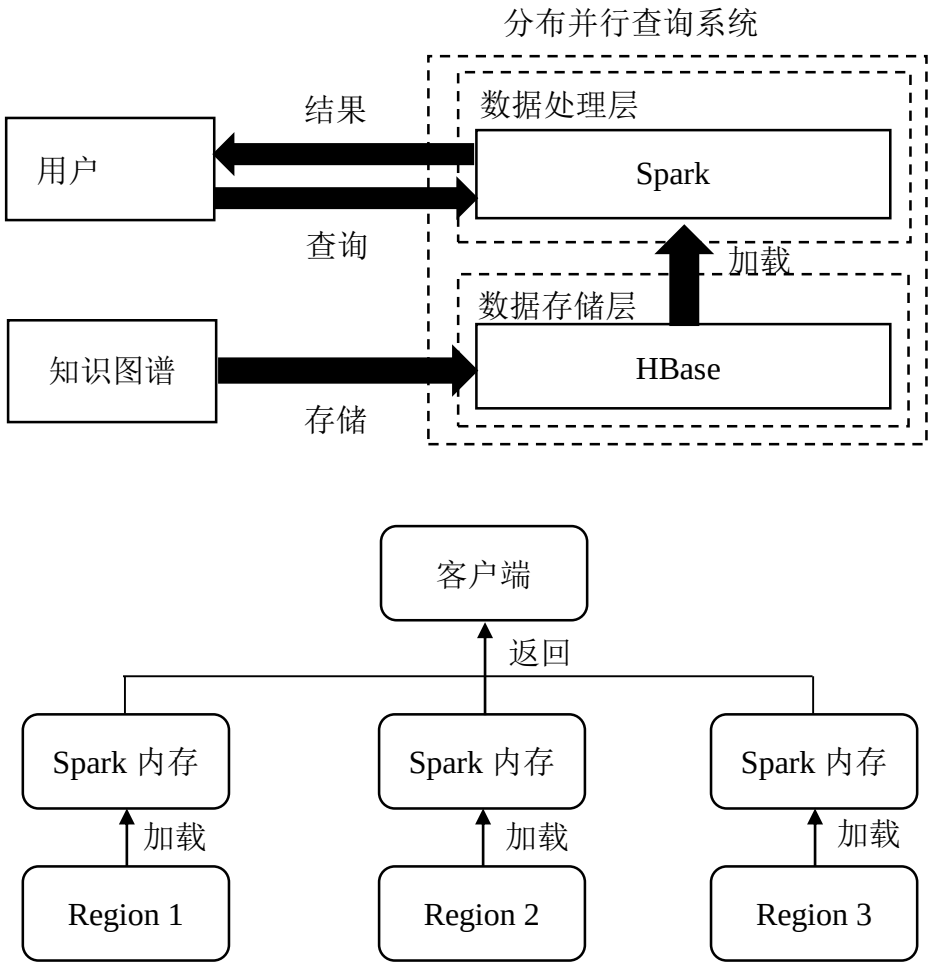


图 4-1 MIQE 设计方案

MIQE 设计方案的具体查询操作过程如下图 4-2 所示。面对基于分布聚集存储模式设计的总体负载均衡，局部聚集存储的知识图谱存储系统，该查询引擎需先将存储系统中的数据通过 newAPIHadoopRDD 的方式加载到 Spark 内存并以抽象集合 RDD 的形式表示，然后 MIQE 对以查询条件生成的查询树自底向上遍历，查询树将会在 5.4 节进行详细说明，本节只需知道查询树的每个节点为实体节点或关系节点。实体节点由多个对同一实体的{属性，属性值}查询条件查询得到的实体结果组成。关系节点由已知查询关系的两个实体组成，根节点属于关系节点。接着 MIQE 判断用户输入的查询语句的查询类型，确定查询类型是单个实体查询，连接查询还是实体关系查询。基于 3.2 节存储模式的设计，若查询条件包含“objects”则表示是连接查询，若查询条件包含“attributes”则表示单个实体查询，实体关系查询需确认返回条件是否出现在查询条件的谓语中。判断查询类型之后，MIQE 根据对应的查询类型使用 Spark 对应的操作算子对该节点并行地查询，查询时并行度为知识图谱存储系统中 Region 的个数，遍历直至到根节点，最后通过 top 算子以有序的方式从集群中将查询结果返回给客户端。

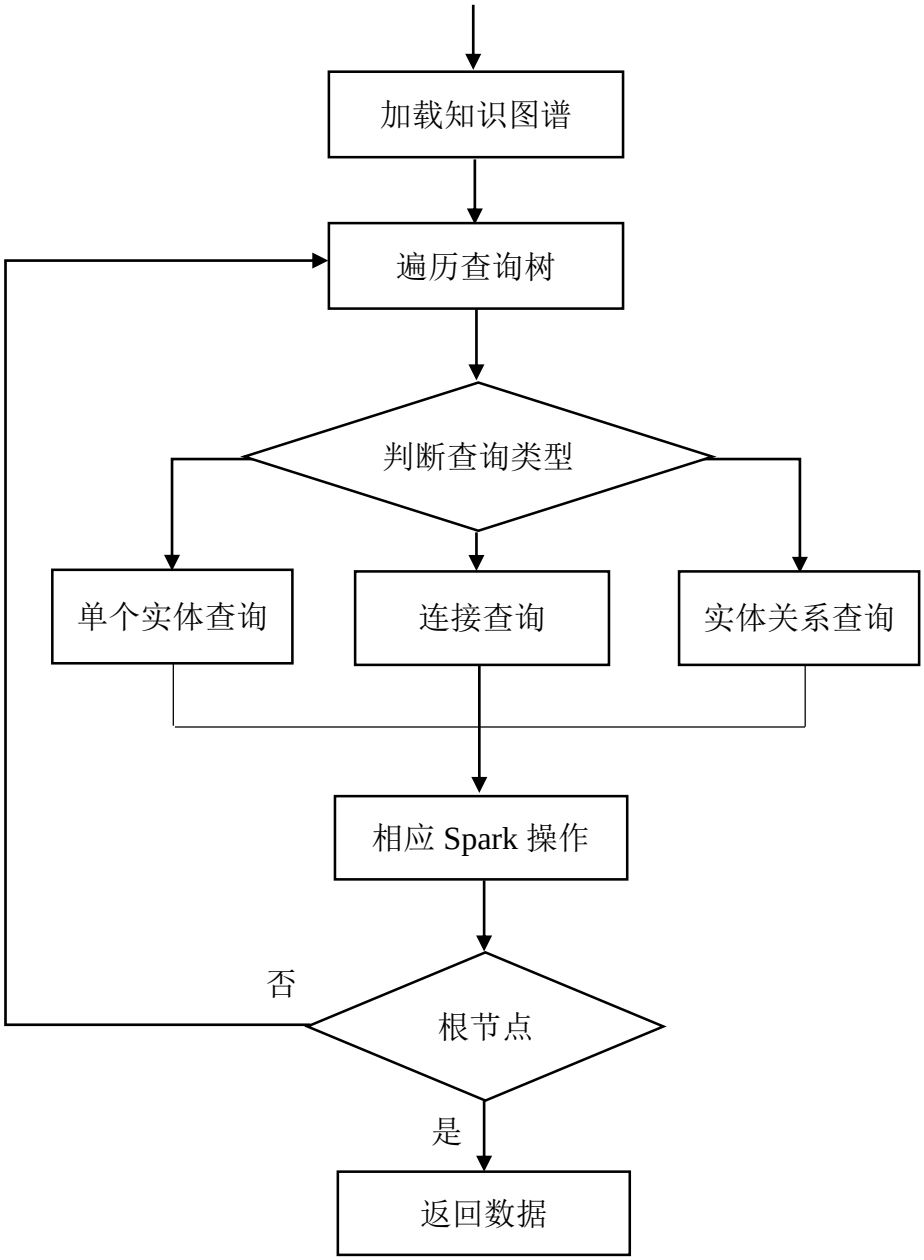


图 4-2 MIQE 设计方案的具体查询操作过程

4.1.1 单个实体查询

MIQE 设计方案的单个实体查询流程如下：对于已加载的知识图谱，查询系统根据查询条件含有“Attributes”判定为单个实体查询后，使用 Spark transformation 中的 filter 算子过滤所有不满足查询条件的实体，最后返回所有符合查询条件的实体。具体伪代码如下算法 4-1：

算法 4-1 基于 MIQE 设计方案的单个实体查询

输入：查询条件
输出：实体对象

1. for 查询条件 do
2. Spark 的 filter 操作；

3. 根据查询条件进行 filter 操作的判断
4. If(实体对象不满足查询条件) Then
5. 返回；
6. else
7. 保存实体对象；
8. 使用 mapToPair 操作将实体对象新建为<实体对象行键，实体对象>对象
9. end for
10. 返回所有新建实体对象;

4.1.2 连接查询

MIQE 设计方案的两个实体间的连接查询的查询树由三个节点构成：查询条件中含有“objects”字符的表示为父节点，即关系节点，其余两个含有“attributes”字符的表示父节点的左孩子节点和右孩子节点，即实体节点。连接查询需自底向上遍历查询树，每个孩子节点分别做基于算法 4-1 的单个实体查询，最后父节点使用 Spark transformation 中的 join 操作算子连接左右孩子的查询结果集，完成一次两个实体间的连接查询。三个实体间的连接查询是将两个实体间的连接查询的结果作为单个实体查询的结果，然后再做一次两个实体间的连接查询。

具体地基于 MIQE 设计方案的连接查询伪代码如下算法 4-2：先对查询树进行后序遍历并初始化标记 Tag，标记 Tag 的作用是判断实体结果集为单个实体查询的结果集还是做完实体间连接操作的结果集。如果 Tag = 0，表示是基于算法 4-1 的单个实体查询的实体集，否则表示是连接操作之后的实体结果集。之后使用 Spark transformation 的 join 算子将实体与实体做连接操作，直至查询条件中的含有“objects”字段的关系均被遍历后结束。

算法 4-2 基于 MIQE 设计方案的连接查询

输入：查询条件
输出：实体对象

1. 对树进行后序遍历，初始化 Tag ← 0;
2. While(含有“objects”字段的关系未被遍历) do
3. If(Tag == 0) then
4. 基于算法 4-1 的单个实体查询，返回实体 1，对象格式<实体 1 row key ，实体 1>;
5. 基于算法 4-1 的单个实体查询，返回实体 2，对象格式<实体 2 row key，实体 2>;
6. Spark 内置的 join 算子，使用 join 算子需保证实体 1 row key = 实体 2 row key，语句形如：返回实体 1 ← 实体 1.join（实体 2）,;
7. Tag ← 1;
8. 根据返回条件，使用 mapToPair 操作将返回实体 1 对象新建为<实体对象行键，返回对象>对象
9. End
10. 返回满足所有条件的实体对象;

4.1.3 实体关系查询

基于 MIQE 设计方案的两个实体的关系查询，需先分别进行单个实体查询，然后在返回的两个结果集的“objects”列簇中遍历列判断是否存在对应的关系，若存在则返回该列的值。多个实体的关系查询则是在两个实体关系查询的基础上再进行同样方法的查询。具体的两个实体关系查询的伪代码如下：先对实体 1 和实体 2 分别进行基于算法 4-1 的单个实体查询，记录返回结果，然后在实体 1 中查看是否存在 objects 列簇，若存在，则遍历该实体 objects 列簇下的所有列值，查询是否有值等于另一个实体 2，若有，则返回相应的关系。

算法 4-3 两个实体的实体关系查询

输入：查询条件
输出：关系

```
1.  基于算法 4-1 的单个实体查询，记录实体 1 集合；
2.  基于算法 4-1 的单个实体查询，记录实体 2 集合；
3.  For 实体 1 集合 do
4.    If(实体 1 存在 objects 列簇) then
5.      For 实体 2 集合 do
6.        If(实体 1 中 objects 列簇中的列值 == 实体 2) then
7.          保存列名，即关系；
8.      End for
9.  End fror
10. 返回所有保存的关系；
```

本节基于上章节设计的分布聚集存储模式设计出采用内存迭代技术的查询引擎 MIQE。该查询引擎不仅通过内存迭代的方式减少磁盘的 I/O，而且由于底层数据具有分布聚集的特性，还可以避免数据倾斜的发生并能对所有 Region 进行并行计算，充分利用集群性能，加快查询速度。从理论上来说，采用内存迭代技术的查询引擎 MIQE 的查询效率会比 Franke 等[6]和 Li 等[17]提出的方案高。但是其也有缺点，如在运算中出现内存溢出的情况，那么溢出的数据会存储到磁盘中，增加磁盘的 I/O，造成查询时间的延长。同时 Spark 的运行效率与并行度有关，若设置较多的 Region，即提高并行度，的确可以在某些程度上加快 Spark 的处理速度，但是其也会造成 HBase 大量的 Region 合并和分割操作，产生 GC，影响系统系统性能。为了验证采用内存迭代技术的查询引擎 MIQE 的查询性能，本文会在第五章对 MIQE 进行查询性能实验。

4.2 采用倒排索引技术的查询引擎 IIQE

本节采用倒排索引技术设计查询引擎 IIQE，其旨在使用索引扫描的方法查询实体减少磁盘的 I/O 同时运用 HBase 集群底层的并行扫描能力加速查询。

HBase 作为面向列的非关系型数据库常被人诟病的特性之一就是无法轻易建立二级索引[38]。为了便于建立二级索引，HBase 在 0.92 版本中引入了协处理器，通过协处理器直接将计算过程放置在 HBase 的服务器端，从而轻易完成建立二级索引，谓词下推和访

问权限控制等等操作[39]。协处理器分为两大类：Observer 和 EndPoint。Observer 的功能与关系型数据库中的触发器类似，在特定的事件发生时调用回调函数，如发生 put 操作时调用相关的回调函数。EndPoint 的功能与关系型数据库中的存储过程类似，用户可以将自定义的操作添加到 HBase 服务端，然后可以通过远程调用相关操作（RPC）使操作在 HBase 服务端执行，如统计表的行数操作[40]。

基于协处理器 Observer 和 EndPoint 的特性，IIQE 查询引擎的查询设计方案：通过对查询实体的属性值和关系对象建立倒排索引并结合 HBase 服务器端的并行查询操作加快查询速度。该方案使用协处理器的 Observer 框架建立倒排索引，借鉴 EndPoint 框架的统计表操作，在 EndPoint 框架下对 Region 进行并行查询。

对于倒排索引的存储，本文设计数据与索引分布在同一张表同一个 Region 的方案。若将数据和索引分为两张表存储，如 Franke 方案，虽然通过索引表的建立可以加速查询实体，但是该设计很可能造成由于数据和索引不在同一个 RegionServer 的跨 RegionServer 的查询，跨 RegionServer 的查询其查询速度由网络的传输状况所决定，当遇到网络拥塞状况时，其查询效率会大幅下降。如果使用数据和索引分布在同一张表同一个 Region 的方案，那么不仅可以充分利用索引的性能加速查询，而且则可以避免网络 I/O 通信，保证知识图谱的读性能。本节基于 3.2 节设计的分布聚集存储模式并结合上述的数据和索引存储在同一个 RegionServer 避免网络 I/O 的思想，设计出一个改进的分布聚集存储模式方案，如下图 4-3 所示。改进的 RDF 三元组存储模式方案对主要增加对 RDF 三元组的宾语，即属性值或者实体建立倒排索引的操作。该方案与 3.2 节设计的分布聚集存储模式方案增加如下改进：

（1）增加 index 列簇。Index 列簇只存放一个 column，该 column 下存放的值为 RDF 三元组的主语 row key。只有索引 row key 才能在 Index 列簇下存储主语 row key，主语 row key 并不存储 Index 列簇。

（2）增加索引 row key。利用 HBase 行键查询的特性，将索引放置在行键可以快速查询到对应的索引。基于 HBase 字典序排序的特性，本文在 3.2 节的 RDF 存储模式的基础上，将索引 row key 的首字段设置为其主语 row key 所属 Region 的起始行键，该设计不仅将主语 row key 和索引 row key 清晰的分割开保证索引 row key 和主语 row key 的完整独立而且还保证索引和主语在同一个 Region，避免跨 RegionServer 的查询，提高查询效率。在 Region 的起始行键后索引 row key 由需构建索引的宾语，列簇，列，分割符“_”和建该索引的主语 row key 构成。索引 row key 只在 index 列簇存储数据。

（3）增加 row key “9999a”。每次对宾语建立索引时，都会将宾语对应的谓语存储到 row key “9999a”中，因此查询时可以通过判断在 row key “9999a”是否存在该谓语来判断宾语是否建立索引。若建立了索引那么就先查询索引 row key 再查询主语 row key，最后得到待查询的实体。若查找行键“9999a”时返回为空，即表示该查询条件中的宾语并未建立索引，那么就进行全表扫描操作。“9999a”只在 attributes 列簇的列中存储表示属性或者关系的谓语，其他列簇均不存储。

查询流程举例如下：假设输入的查询条件如下：“?s attributes:NAME Jack”。那么在改进的分布聚集存储模式下首先去查询行键“9999a”，查看列簇 attributes 下的列是否存在属性 NAME，若存在属性 NAME，则对所有 Region 上的索引进行并行范围搜索，索引起始地址：“Region 的起始行键|Jack|attributes|NAME”，索引的终止地址：“Region 的起始行键|Jack|attributes|NAME`”，使用“`”的原因在于“`”的 ASCII 值较大，在以字典序排序的 HBase 中会被最后检索。对查询到的符合条件的所有索引 row key 再根据分

割符“_”进行分割得到主语 row key，最后根据主语 row key 查询目标实体并返回结果。基于改进的的分布聚集存储模式在 Region 分割时，需要以该 Region 的主语 row key 的中位数进行分割且分割完之后需要根据新的 Region 起始行键重建索引。

Row key	attributes		objects		index
	属性	…	关系	…	Column1
0000|索引值 1|cf|column_主语 row key1					主语 row key1
0000|索引值 2|cf|column_主语 row key2					主语 row key2
…					…
0001:类型:实体 1（主语 row key1）	实体属性值（索引值 1）	…	…	…	
0999:类型:实体 2（主语 row key2）	…	…	实体（索引值 2）	…	
1000|索引值 3|cf|column_主语 row key3					主语 row key3
…					…
1001:类型:实体 3（主语 row key3）	实体属性值（索引值 3）	…	…	…	
...	…	…	…	…	…
9000|索引值 4|cf|column_主语 row key4					主语 row key4
…	…	…	…	…	…
9999:类型:实体 4（主语 row key4）	…	…	实体（索引值 4）	…	
9999a		添加索引的属性			

图 4-3 改进的分布聚集存储模式

基于上述改进的分布聚集存储模式，查询引擎 IIQE 使用协处理器的 Observer 框架建立倒排索引。与用 HBase 的客户端 put 操作建立索引不同，通过 Observer 框架在服务器端建立索引不仅可以充分保证数据的完整性和一致性还可以避免索引的丢失。Observer 框架为客户端提供 RegionObserver 接口，旨在便于客户端对数据的操作。Observer 框架不需要客户端代码，其只在特定操作发生时（如 postPut 操作）触发服务端代码实现。本文通过 RegionObserver 接口建立倒排索引的方法如下图 4-4 所示。首先客户端发出 put 请求，然后该 put 请求被分派给合适的 RegionServer 和 Region，Region 产生的结果被 RegionObserver 的 coprocessorHost（协处理器主机）所拦截，拦截后服务器调用 postPut（Put 之后的操作）方法，在 postPut 方法中建立倒排索引，最后将建完倒排索引的信号返回到客户端，若建立索引失败则返回索引失败信号，告知系统重新再建立索引。

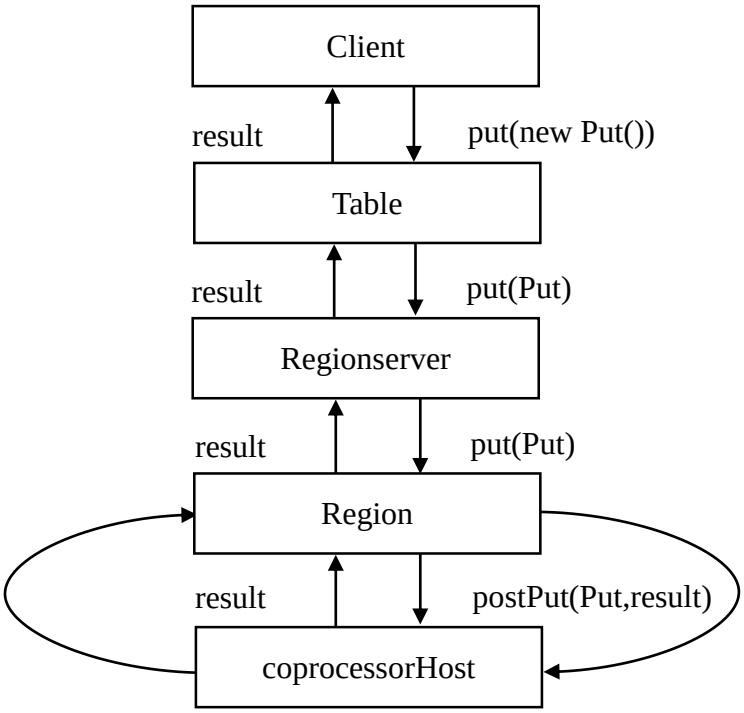


图 4-4 倒排索引建立过程示意图

协处理器的 EndPoint 框架是基于 HBase 服务器端和客户端的的归并扫描技术，其通过 protocl（协议）定义接口实现客户端代码并通过 RPC（Remote Procedure Call，远程过程调用）通信进行数据的搜集归并。例如求表的行数，EndPoint 先在多台存有该表数据的 RegionServer 上同步计数，然后再将每个 RegionServer 的计数值返回给客户端，最后客户端只需进行一次进行全局加和操作即可得表中的行数。EndPoint 的并行计数过程如下图 4-5。用户需要先将自己定义的计数操作添加到服务器端。在执行计数操作时，用户在客户端调用 HBase 的 coprocessorService ()方法，然后 execCoprocessor()方法内的 RPC 操作可以并行对表的每个 RegionServer 的 Region 进行计数操作。

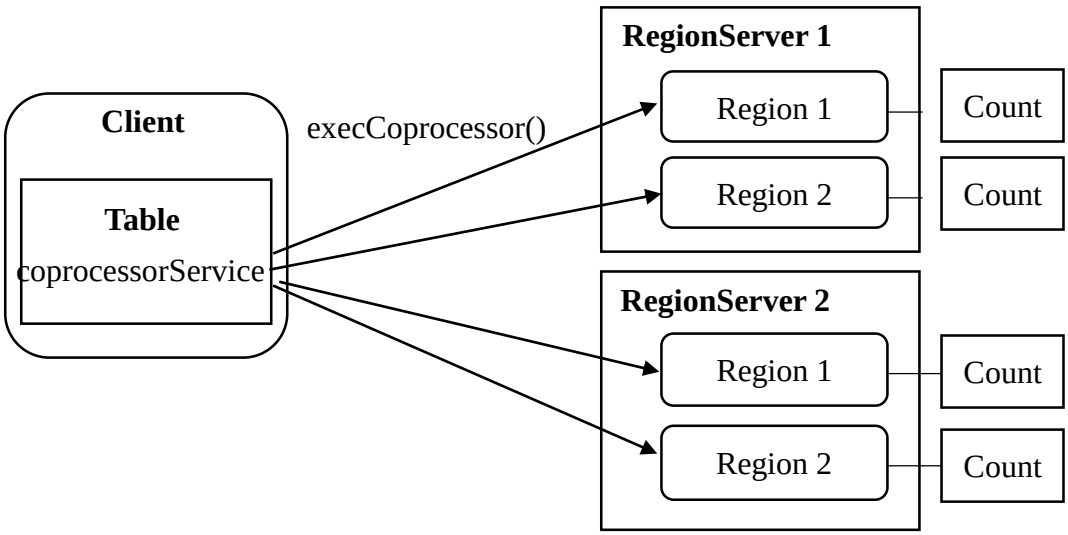


图 4-5 分布并行求表行数示意图

受到使用 EndPoint 框架计数的启发，查询引擎 IIQE 采用协处理器的 EndPoint 框架设计并实现并行查询实体的功能。IIQE 首先定义单个实体查询，连接查询和实体关系查询的查询方法，然后将查询方法的文件配置到服务器端，查询时 EndPoint 框架便会对表的所有 Region 进行对应查询条件的并行扫描，最后扫描结果通过 RPC 通信返回到客户端。使用 EndPoint 框架查询实体的流程与计数类似，只是增加了实现方法的难度。以 Observer 和 EndPoint 协处理器框架实现倒排索引和并行查询的查询引擎 IIQE 的设计方案如下图 4-6。该方案在存储 RDF 三元组时，每存储完一个 RDF 三元组后使用 Observer 协处理器框架对 RDF 三元组的宾语建立倒排索引。在用户查询 RDF 三元组时，使用 EndPoint 协处理器框架在 Region 中并行查询输入的查询条件，通过对 Region 索引的遍历查找到对应的查询实体，若在索引中没有查找到，则对该 Region 数据 row key 部分全表扫描，最后通过 RPC 操作将结果归并到客户端。

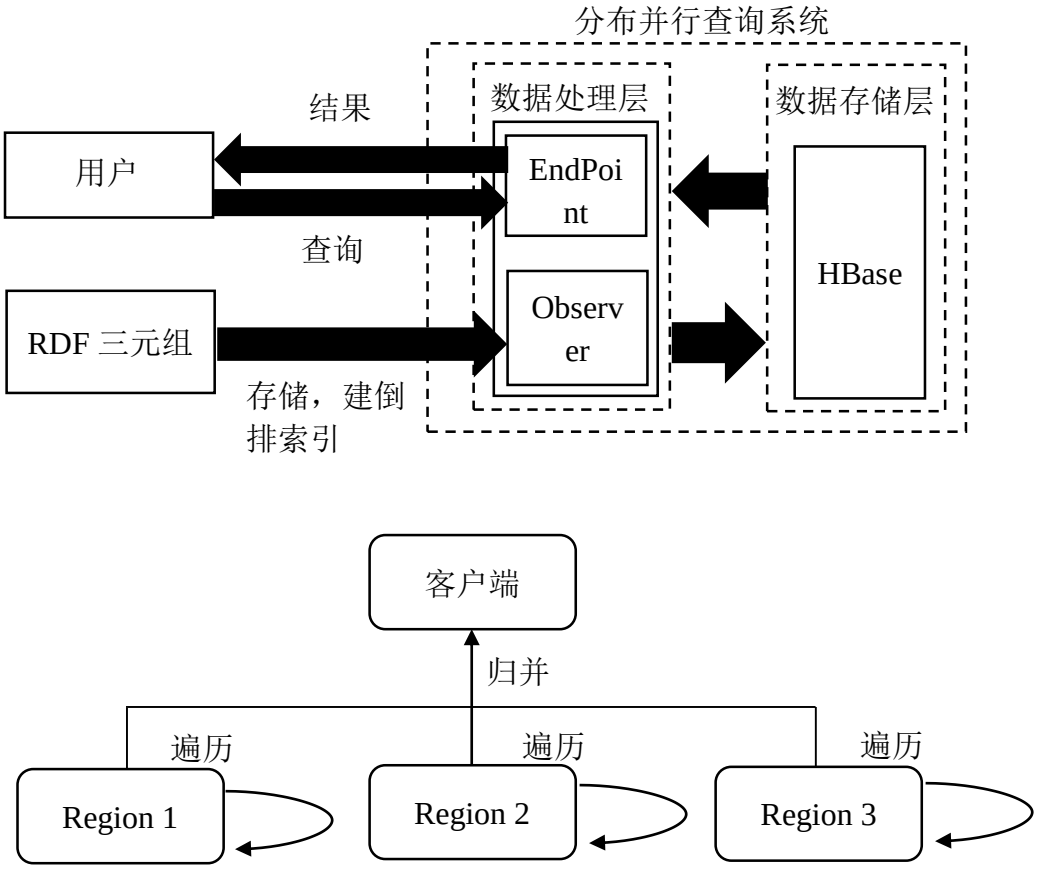


图 4-6 IIQE 设计方案

IIQE 设计方案的具体查询操作过程如下图 4-7 所示，该图与图 4-2 有三处不同：1）不需要事先加载知识图谱数据。2）相对应的 Spark 操作换成协处理器操作。3）添加客户端操作，由于 IIQE 的并行查询均在各个 Region 上执行，对于多个{属性-属性值}条件查询的交操作和连接查询的连接操作需要放置在客户端进行。本节设计的改进的分布聚集存储模式在添加了索引的基础上依旧保持着总体负载均衡，局部聚集存储的特性，因此查询引擎 IIQE 不仅通过使用倒排索引减少磁盘的 I/O，而且充分利用 Region 的并行能力加快查询速度，还保持知识图谱良好的负载均衡状态。

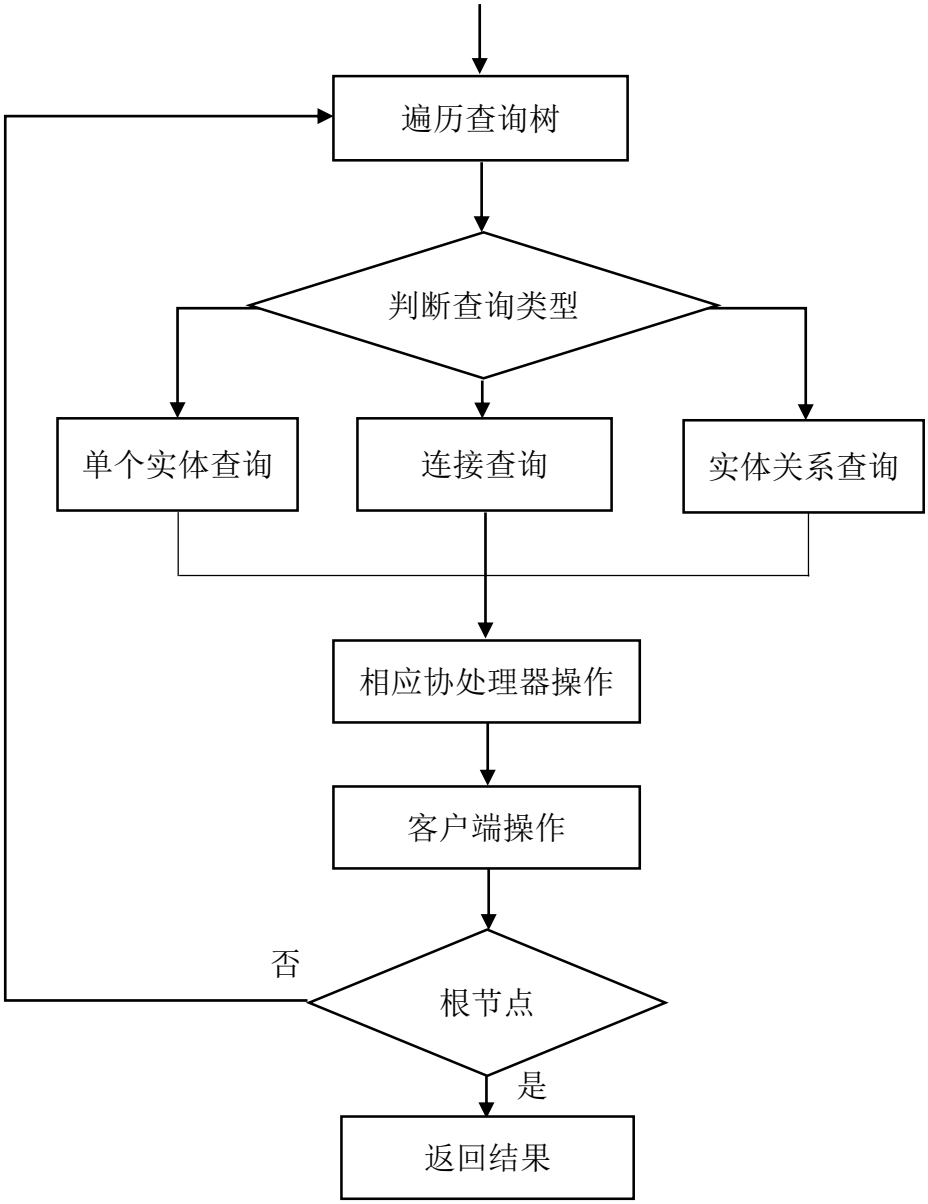


图 4-7 IIQE 设计方案的具体查询操作过程

4.2.1 单个实体查询

IIQE 的单个实体查询方案与 MIQE 的单个实体查询方案相比较为复杂。因为实际操作中并不会对所有的谓语建立索引，因此在查询时，需要判断叶子节点中的属性值或者关系实体是否建立了索引，若建立了索引，那么通过倒排索引查找实体，若没有，则需使用 HBase 的过滤器并行遍历所有的 Region。若有多个查询条件最后还需要对所有返回的实体在客户端做数据的交操作。基于 IIQE 设计方案的单个实体查询伪代码如下：遍历所有查询条件，然后在 HBase 中查询行键“9999a”，查询是否存在查询条件中的谓语，若存在，那么根据索引行键的格式：“Region 起始行键|value|column_主语 rowkey”进行查询，然后对索引行键分割，获取主语 rowkey，最后根据主语 rowkey 查询到对应的实体。若没有建立索引，那么使用 HBase 的 SingleColumnValueFilter 过滤器根据查询

条件在每个 Region 上遍历，遍历完所有查询条件后再在客户端对返回所有实体做交操作，最后返回最终结果。

算法 4-4 基于 IIQE 设计方案的单个实体查询

输入：查询条件
输出：实体对象

```
for 查询条件 do
  If(hbase 中的 9999a 行不包含查询条件中的谓语) then
    遍历 HBase 全表，使用 SingleColumnValueFilter 方法查询；
  Else
    通过协处理器在所有的 Region 中根据查询条件范围查询倒排索引；
    对索引进行分割，分割得到主语 row key；
    根据主语 row key 使用 HBase 的 GET 方法查询得到实体对象，返回客户端保存；
end for
在客户端对所有得到的实体对象进行交操作；
返回满足查询条件的所有实体对象；
```

4.2.2 连接查询

基于 IIQE 设计方案的的连接查询与基于 MIQE 设计方案的的连接查询类似。只是由于 HBase 本身没有连接方法，所以需要设计连接操作。IIQE 使用 Hash Join 的方法进行实体间的连接操作。Hash Join 就是将两张表中的一张表作为 hash 表，然后去扫描另一张表，扫描过程中去查询是否有与 hash 表满足等值条件的内容。在结果集很大的情况，使用 Hash join 比使用 nested loop join，即嵌套循环的查询方式更加有效率。对于一张表的 Hash Join 本文使用第一个单个实体查询的结果作为 hash 表，然后让第二个实体查询的结果与第一个实体查询的结果进行比较，以此类推，最终获得所有满足连接查询条件的实体。基于 HBase 协处理器查询方案的连接查询伪代码如算法 4-5 所示，为了做 Hash Join 的操作需要将实体 1 保存为<实体 1 宾语值，实体 1 对象>的格式，实体 2 保存为<实体 2 主语值，实体 2 对象>的格式，之后通过判断实体 2 主语值与实体 1 宾语值是否相等做 Hash Join。

算法 4-5 基于 IIQE 设计方案的连接查询

输入：查询条件
输出：实体对象

```
对树进行后序遍历，初始化 Tag ← 0；
While(关系未被遍历) do
  If(Tag == 0) then
    基于算法 4-4 的单个实体查询，返回实体 1 所有对象，返回格式
    为<实体 1 宾语值，实体 1 对象>；
  基于算法 4-4 的单个实体查询，返回实体 2 所有对象；返回格式为
  <实体 2 主语值，实体 2 对象>；
  If(实体 2 主语值 == 实体 1 宾语值) then
    保存实体；
```

10. Tag ← 1；
11. End
12. 返回满足所有条件的实体对象；

至于实体关系查询，基于 IIQE 设计方案的实体关系查询的查询流程与基于 MIQE 设计方案的实体关系查询流程一致，只需将单个实体查询方法换成算法 4-4 的单个实体查询算法即可，因此本节不再赘述。

本节设计的查询引擎 IIQE 通过倒排索引技术减少磁盘的读写次数，加速查询，同时为了存储索引改进了第三章设计的分布聚集存储模式，在基于原分布存储模式的下设计将数据和其对应的索引存储在同一个 Region 的方案，改进的存储模式可以避免查询时出现的跨 RegionServer 的网络 I/O 通信，从而再次加速查询，最后利用 HBase 协处理器 EndPoint 框架并行计算的特性，在查询时对集群中拥有该表的所有 Region 进行并行查询操作，最大化加速查询。理论上，采用倒排索引技术的查询引擎 IIQE 的查询效率与查询引擎 MIQE 一样会比 Franke 等[6]和 Li 等[17]提出的方案高。但是其也有缺点，如数据交操作和 Hash Join 在客户端运行，因此交操作和 Hash Join 的性能将会受客户端机器性能的影响，同时由于官方推出协处理器的时间较晚，目前对协处理器进行并行查询的性能并没有相关资料可以参考。为了验证采用倒排索引技术的查询引擎 IIQE 的查询性能，本文会在第五章对 IIQE 进行查询性能实验。

4.3 本章小结

基于第三章设计的分布聚集存储模式，本章设计出两种采用不同方案的分布并行查询引擎：MIQE 和 IIQE。MIQE 采用分布式内存迭代技术以过滤和连接操作并行查询抽象集合表示的实体，IIQE 采用倒排索引技术并利用集群分布并行扫描能力以归并的方式查询实体。上述两种分布并行查询都旨在通过减少磁盘 I/O 消耗和最大化并行查询的方式保证知识图谱查询可扩展，提升知识图谱系统的读性能，加快知识图谱查询速度。本文将会在第五章对 MIQE 和 IIQE 进行查询性能的验证和比较。

第五章 原型系统的实现和性能验证

本章设计并实现基于大数据平台的知识图谱存储访问系统。该系统由数据载入模块，SPARQL 语句解析模块和数据查询模块三大模块完成知识图谱的存储查询功能。实验验证，基于分布聚集存储模式和分布并行查询引擎实现的知识图谱存储访问系统具有良好的水平扩展性，面对大规模知识图谱，IIQE 查询引擎的查询性能比 MIQE 查询引擎和 Franke 查询引擎性能更加优异。

5.1 知识图谱存储访问系统的系统结构

本章设计的基于大数据平台的知识图谱存储访问系统由三个模块组成：数据载入模块，SPARQL 语句解析模块和数据查询模块。数据载入模块的主要功能是将不同数据来源的 RDF 三元组集合（如 MySQL，.nt 文件）以设计的存储模式导入到 HBase 中。SPARQL 语句解析模块的主要功能是对 RDF 的蕴含语义信息的查询语句 SPARQL 进行解析，获取其查询结构。数据查询模块的主要功能是对 SPARQL 语句解析模块解析出的查询结构构建查询树，然后再以设计的查询引擎（如：MIQE，IIQE）对存储的知识图谱进行实体或者关系的查询。本章设计的知识图谱存储访问系统遵循软件工程高内聚低耦合的设计原则，每个模块只完成系统要求的独立子功能并与其他模块的联系较少且接口简单。完整的基于大数据平台的知识图谱存储访问系统的系统结构图如下图 5-1 所示。虽然上述实验均使用 N-Triple 格式表示 RDF 三元组的.nt 文件，但是考虑到现实生产环境中大部分数据都存放于 MySQL 等数据库，因此本系统在数据载入模块不仅设计了.nt 文件导入知识图谱到 HBase 的存储方案还设计了 MySQL 数据库导入知识图谱到 HBase 的方案。对于上述两种数据来源的知识图谱，数据载入模块提供统一的知识图谱数据导入接口，未来当有其他的数据来源时，只需实现对应接口即可，不需修改底层数据载入框架，利于系统存储方式的扩展。由于 SPARQL 语句本身不支持 HBase 的查询操作，因此在查询时需要对用户输入的 SPARQL 查询语句进行解析来获取查询结构为下一步查询树的生成做基础。本文在 SPARQL 语句解析模块使用 Antlr[41]对 SPARQL 语句进行解析。Antlr 是一款优秀的开源语法解析器和生成器，经常用于对已给定词法规则和语法规则的语言进行识别。给定 SPARQL 语法规则，本文通过使用 Antlr 自动解析得到用户输入 SPARQL 语句的前缀、返回条件和查询条件等等信息。基于获取的查询结构，知识图谱存储访问系统在数据查询模块构建查询树，对于构建的查询树，系统可以使用不同的查询引擎进行查询，如采用内存迭代技术的查询引擎 MIQE 或者采用倒排索引技术的查询引擎 IIQE 等，与数据载入模块一样，数据查询模块也提供了统一的查询接口，利于系统查询方式的扩展。

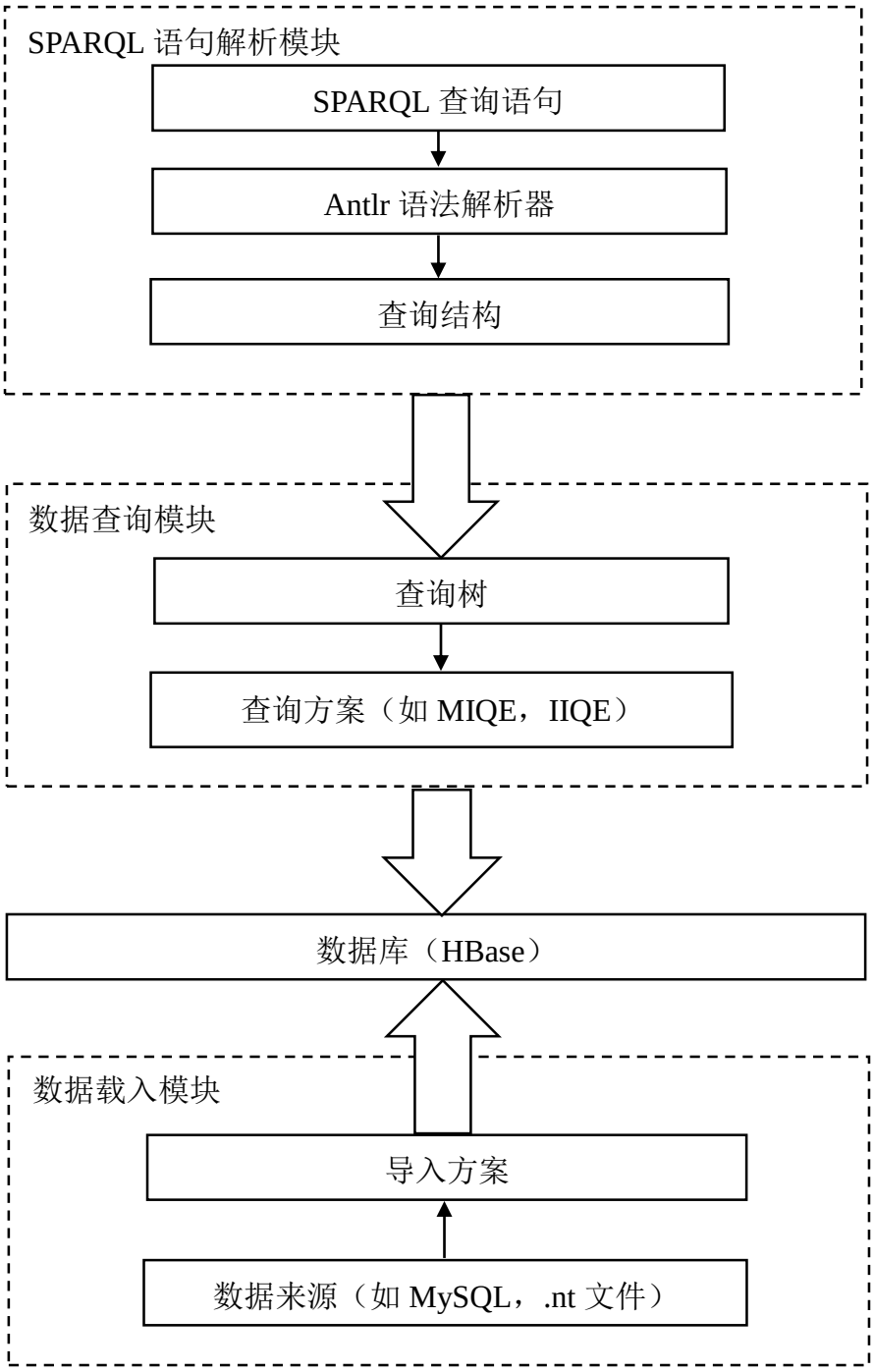


图 5-1 基于大数据平台的知识图谱存储访问系统的系统结构图

5.2 知识图谱数据载入模块的设计与实现

5.2.1 基于关系型数据库的数据载入方案

在关系型数据库中，知识图谱的实体和实体间的关系主要由关系表中的行和关系表中外键所对应的列表示。在遍历关系型数据库的数据时，可以通过表的主键和外键信息寻找每行对应的外键行，将外键对应的列名作为关系名，对应的外键行作为对应关系的实体。基于关系型数据库的实体构建示意图如下图 5-2 所示。主键 1 对应的行表示实体

1，主键 2 对应的行表示实体 2，外键 1 对应的行表示实体 3。由于主键 1 在列名 N 在存在外键 1，那么实体 1 与实体 3 之间存在关系列名 N。除此之外实体 1 包含属性 XXXX，实体 2 包含属性 YYYY，实体 3 包含属性 ZZZZ。

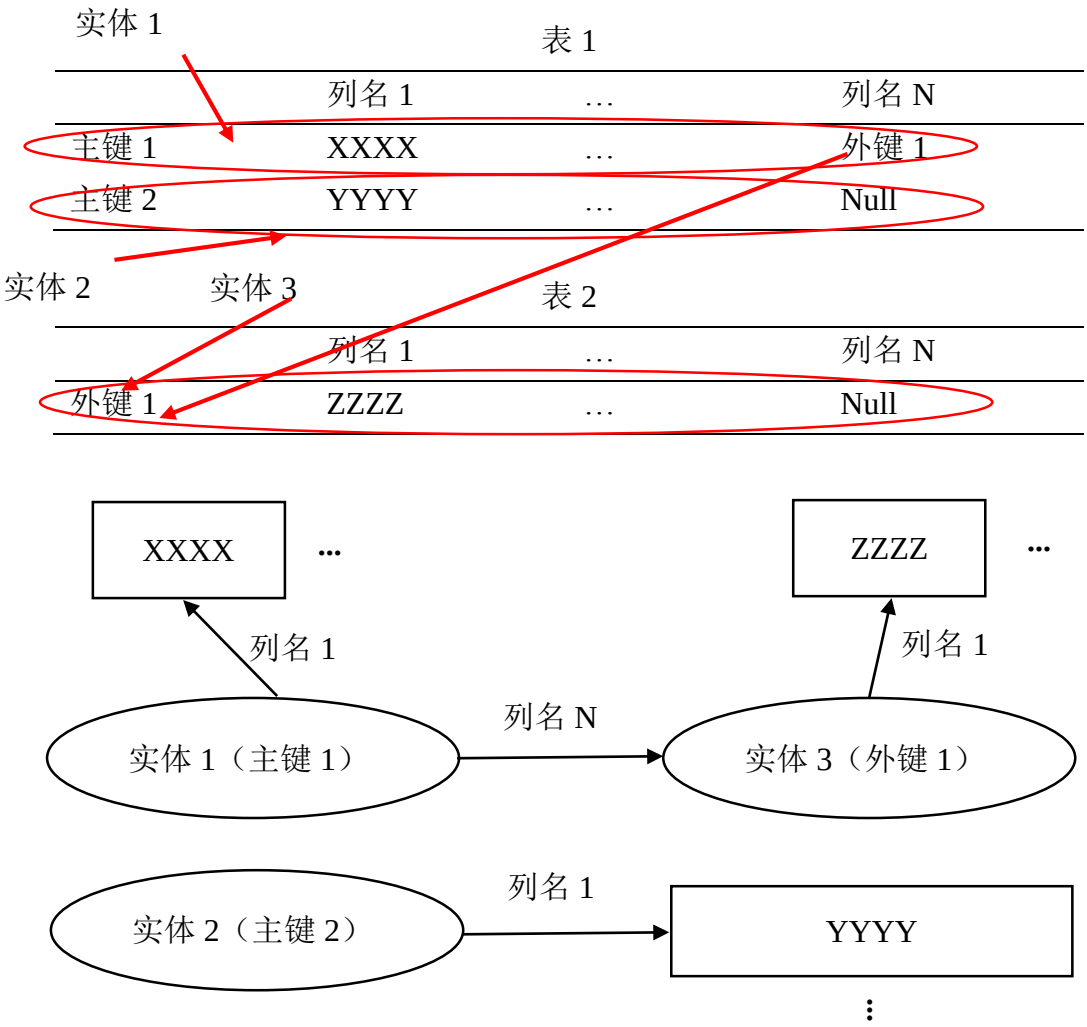


图 5-2 实体构建示意图

将关系型数据库的数据载入到 HBase 的流程如下图 5-3 所示。首先需要获取数据库中所有表的表名和对应的主键信息和外键信息，然后遍历每个表的每行，针对每行还需要遍历该行的每一列，判断该列是否为外键，若是，那么根据外键信息，获取外键关联的表，对于外键关联的表根据外键构建查询，获取关联的表所对应的关联行并通过关联行主键的信息，构建实体的关系，保存为<列名，关联行主键>的形式。若不是，那该列为实体的属性值，保存为<列名，属性值>。数据库的每一行构建为一个实体对象，其包含普通的属性和实体间的关系这两类信息。最后将对象写入 HBase，若表未读完则继续遍历。基于图 5-3 的流程，MySQL 构建实体导入 HBase 如算法 5-1 所示。

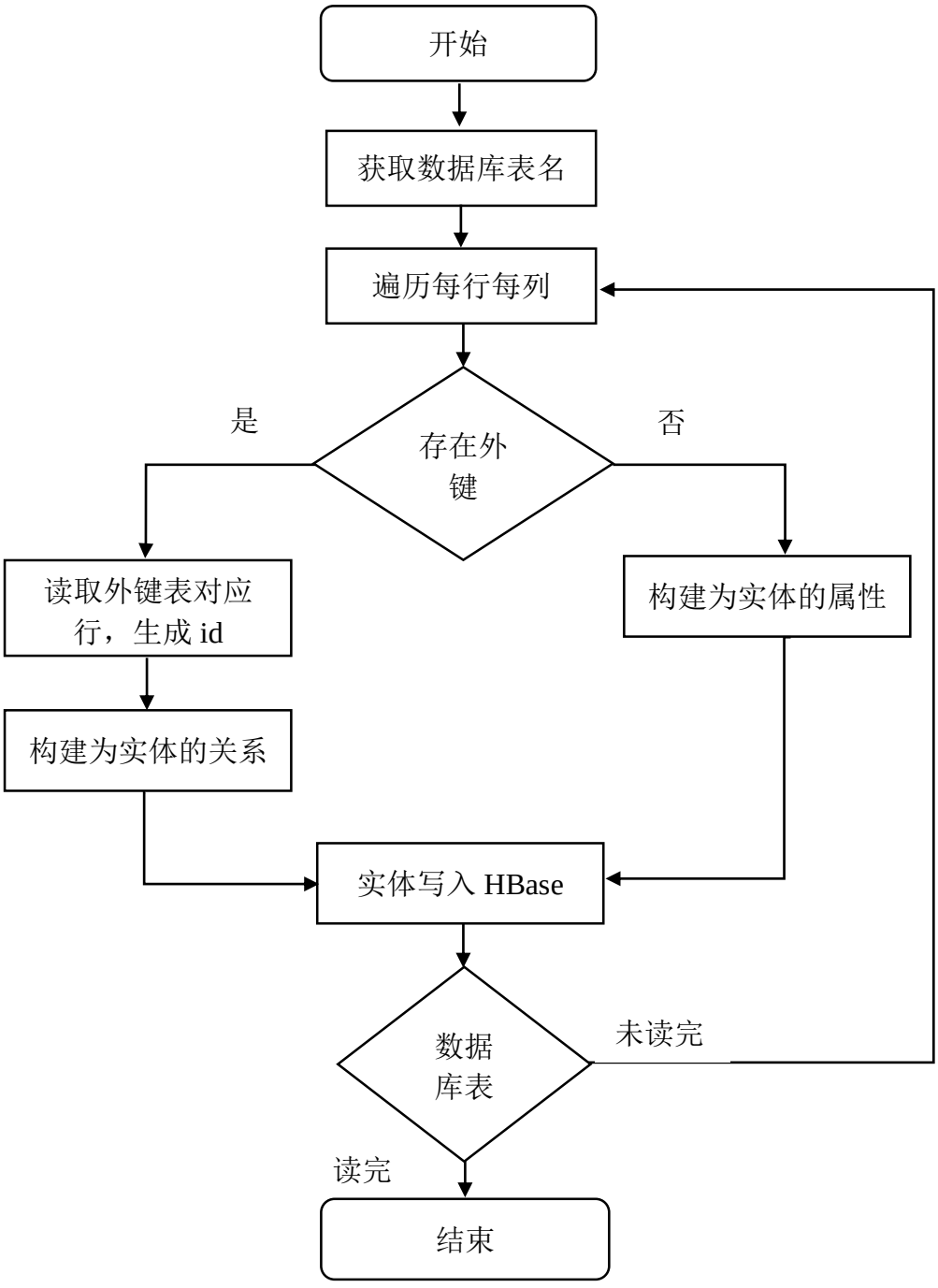


图 5-3 基于关系型数据库数据载入方案流程图

算法 5-1 MySQL 构建实体导入 HBase

输入：数据库连接信息
输出：实体对象集合，存入到 HBase

```
1.    for 每张表 do
2.        for 每一行 do
3.            for 每一列 do
4.                if 列名 ∈ 数据库连接信息中的外键信息 Then
```

```
5.                    使用 sql 语句查询外键所对应的表 id;
6.                    构建为实体的关系;
7.                Else
8.                    构建为实体的属性;
9.            end for
10.       end for
11.   end for
12.   批量 put 实体对象到 HBase;
```

5.2.2 基于.nt 文件的数据载入方案

本文使用 N-Triple 格式存储 RDF 三元组，N-Triple 格式已在 2.1.1 节进行了介绍，本节不再赘述。以 N-Triple 格式表示的 RDF 三元组集合常以.nt 文件存储。使用.nt 文件构建实体主要有两大难点：1）N-Triple 中的重复出现的主语。2）实体与实体之间关系的存储。由于 HBase 中的 row key 唯一，因此对于重复主语的问题，将主语存储为 row key 即可解决。对于存储实体与实体之间关系的问题，通常有两种方案可以解决。第一种为实体与实体之间关系不多且已知的情形，其只需在遍历.nt 文件时，对遇到对应为关系的字符存储为关系其余则存储为属性即可。第二种为实体与实体之间的关系多且未知的情形，那么需要使用深度优先遍历搜索每一个 N-Triple 格式的 RDF 三元组的宾语，观察这一个 N-Triple 格式的宾语是否会是其他 RDF 三元组的主语，若是，则该 RDF 三元组中的谓语和宾语以<谓语，宾语>的形式存储为关系，若不是，则以<谓语，宾语>的形式存储为属性。由于本文的数据集来自 LUBM，其关系情况未知，因此本文使用第二种方案构建实体并将实体导入到 HBase，具体流程如下图 5-4。对于深度优先遍历搜索，本文使用栈保存 RDF 三元组。扫描每一行 RDF 三元组时，先判断该 RDF 三元组是否为文件中的最后一个，若是，那么判断栈是否为空，若栈为空，则遍历结束。若栈不为空，则栈中保存的 RDF 三元组出栈，对该出栈的 RDF 三元组判断其宾语是否为其他 RDF 三元组的主语，若是，确定出栈的 RDF 三元组的谓语为关系，对该 RDF 三元组以对应关系的形式存储在 HBase 中，并将在判断中得到的对应 RDF 三元组入栈，重复流程。若不是，确定出栈的 RDF 三元组的谓语为属性，对该 RDF 三元组以对应属性的形式存储在 HBase 中，遍历下一个 RDF 三元组，重复流程。若扫描的 RDF 三元组不是文件的最后一个 RDF 三元组，则需判断栈是否为空，若栈为空则对该 RDF 三元组判断其宾语是否为其他 RDF 三元组的主语，过程如前所述。若不为空，则栈中的 RDF 三元组出栈，然后对出栈的 RDF 三元组判断其宾语是否为其他 RDF 三元组的主语，过程如前所述。

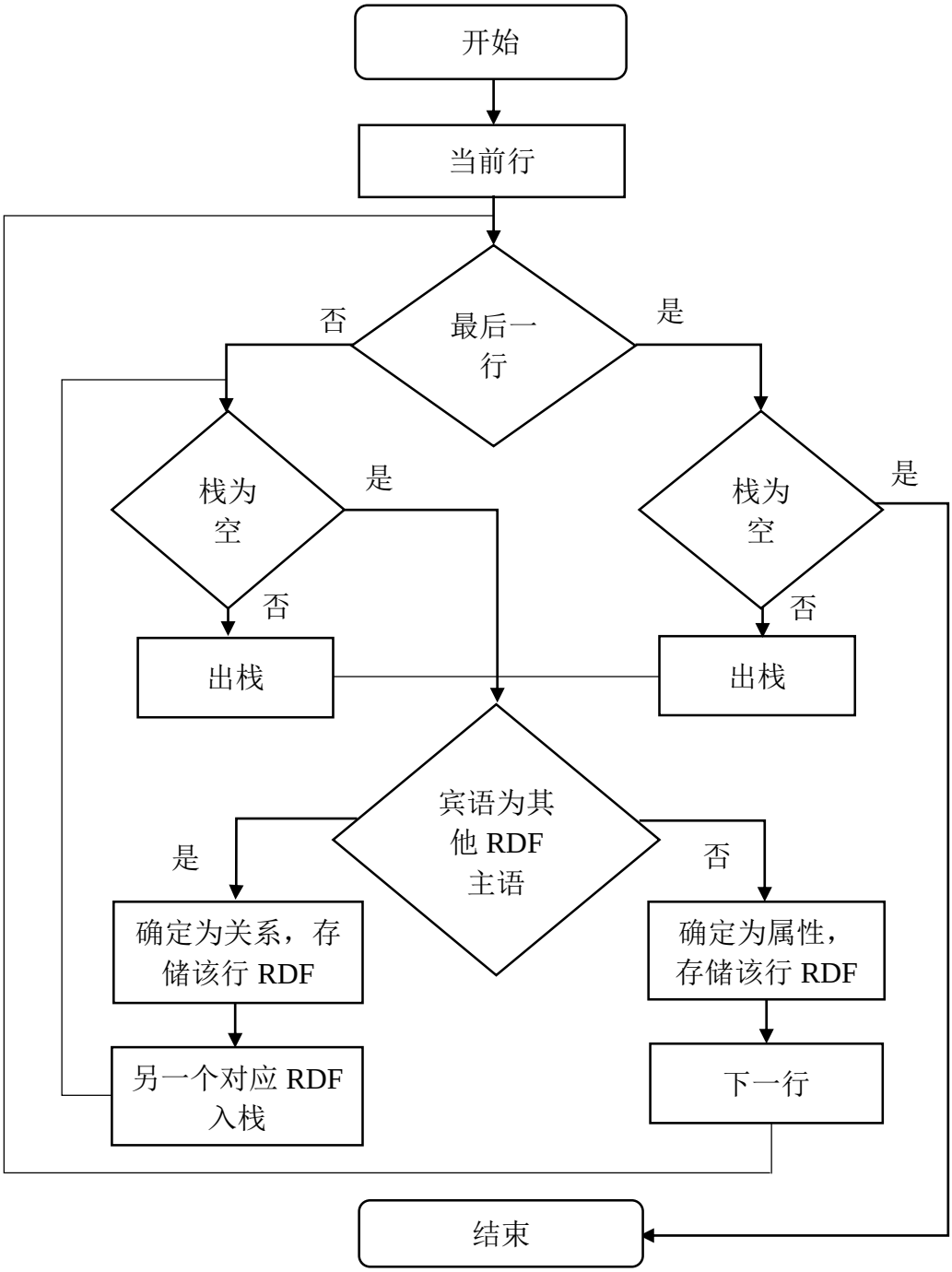


图 5-4 基于.nt 文件的数据载入方案流程图

5.3 知识图谱 SPARQL 语句解析模块的设计与实现

由于面向 RDF 图的 SPARQL 查询语句不支持 HBase 相关查询操作，所以在使用 SPARQL 查询以 HBase 存储的知识图谱时需要对 SPARQL 查询语句进行解析。语句解析一般分为两个步骤：词法分析和语法分析。词法分析是将字符聚集为单词或者符号的过程。语法分析则是在词法分析的基础上将单词或者符号序列组合成已给定语法规则的语法短语。如下图 5-5 的针对赋值语句“s=100;”的解析，机器在识别的过程中先通过

词法分析器分析出每个词，然后再通过语法分析器，根据先前设定的语法确定其为赋值语句，最后结果可以通过语法分析树展示。

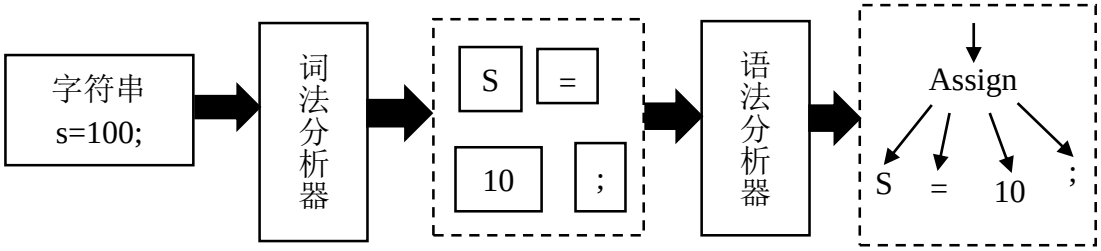


图 5-5 语句解析过程图

SPARQL 查询语句解析与上述赋值语句解析类似，具体流程如图 5-6 所示，对于每个输入的 SPARQL 语句先进行 SPARQL 词法分析获取字符流，基于字符流再进行 SPARQL 语法分析获取解析树，对解析树进行遍历即可获取输入的 SPARQL 语句的查询结构，如前缀、查询条件和返回条件等等，获取 SPARQL 查询结构后可以在系统数据查询模块构建查询树，然后以 HBase 的查询方式进行查询。

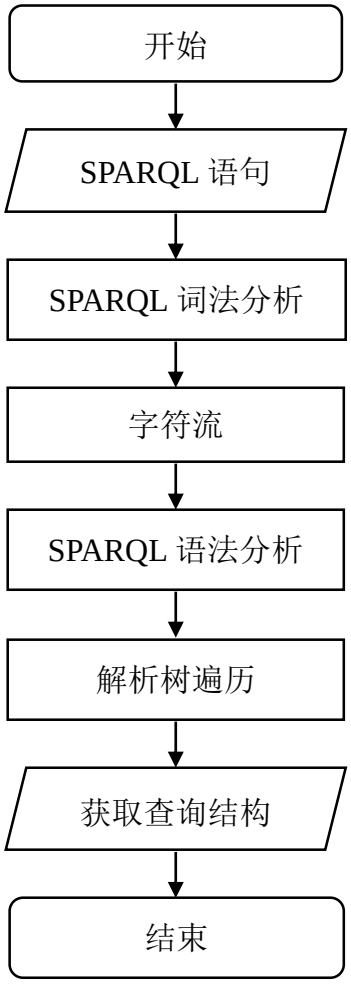


图 5-6 SPARQL 语句解析流程图

鉴于个人设计并实现 SPARQL 语法解析器难度极大，因此，对于 SPARQL 查询语句的解析，本文使用的是第三方开源语法解析器工具 Antlr，Antlr 解析 SPARQL 过程如下：先基于 Antlr 语法编写满足 SPARQL 规则的语法规则文件，之后每当输入一个 SPARQL 查询语句，Antlr 会根据该语法规则文件自动进行词法分析、语法分析，生成相应的解析树，通过遍历解析树，数据查询模块便可以获取到对应 SPARQL 语句的查询结构。举例如下图 5-7 所示，输入一个含有两个查询条件的 SPARQL 查询语句，通过 Antlr 解析后 SPARQL 语句解析模块可获得对应查询语句的查询结构，其中“query”表示输入语句，“PREFIX”表示前缀，“selectQuery”表示查询语句，“EOF”表示终止符，“TripleBlocks”表示三元组集合，“TripleBlock”表示三元组。

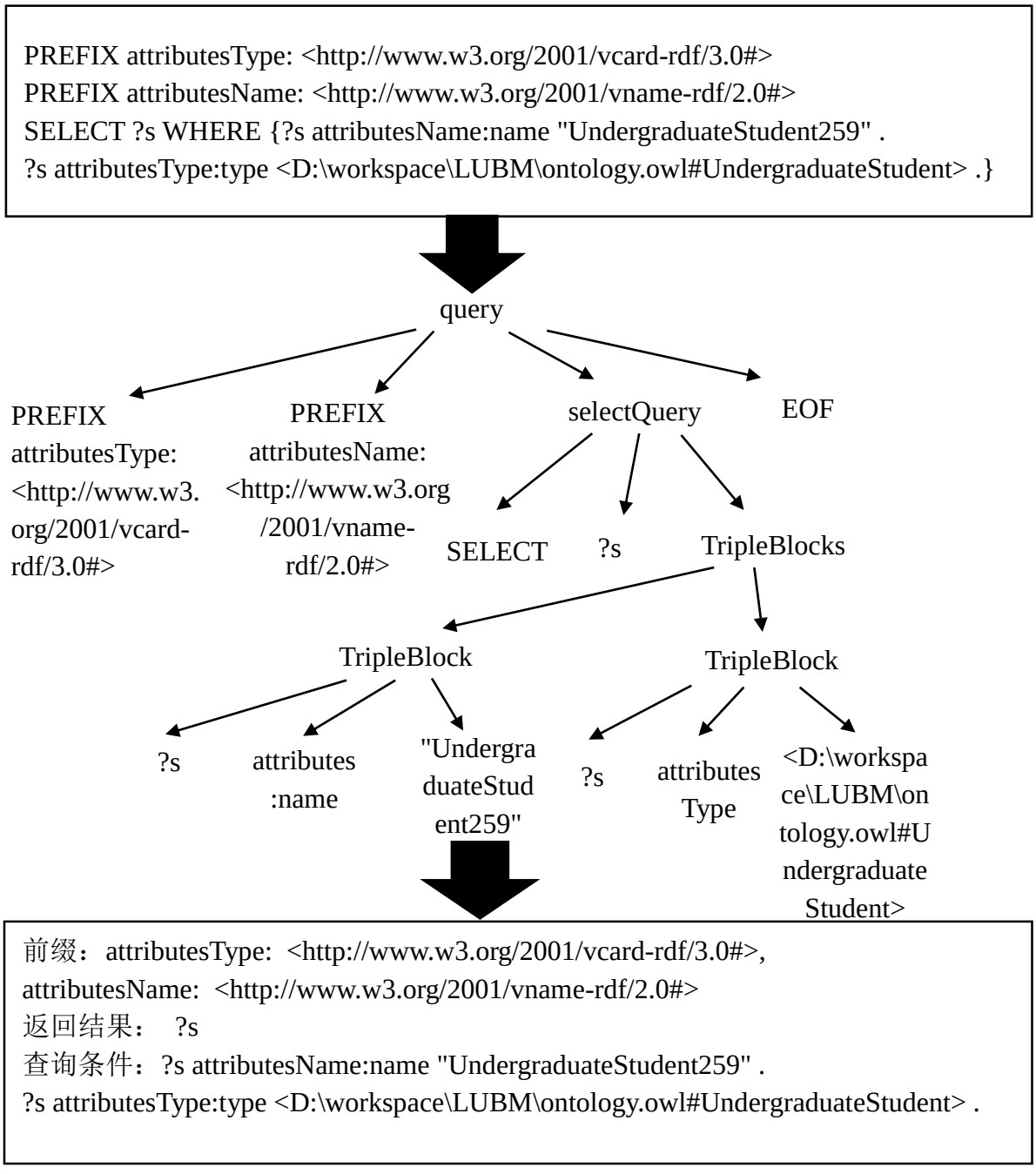


图 5-7 SPARQL 语句解析过程图

5.4 知识图谱数据查询模块的设计与实现

5.4.1 二叉查询树的构建

在完成 SPARQL 查询语句的解析后，基于获得的查询结构，系统在数据查询模块递归构建二叉查询树。二叉查询树结构如下图 5-8 所示，树的节点一共分为两种：实体节点和关系节点。实体节点由多个对同一实体的{属性，属性值}查询条件查询得到的实体结果组成。关系节点由已知查询关系的两个实体组成，根节点属于关系节点。二叉查询树的构建算法如算法 5-2 所示。

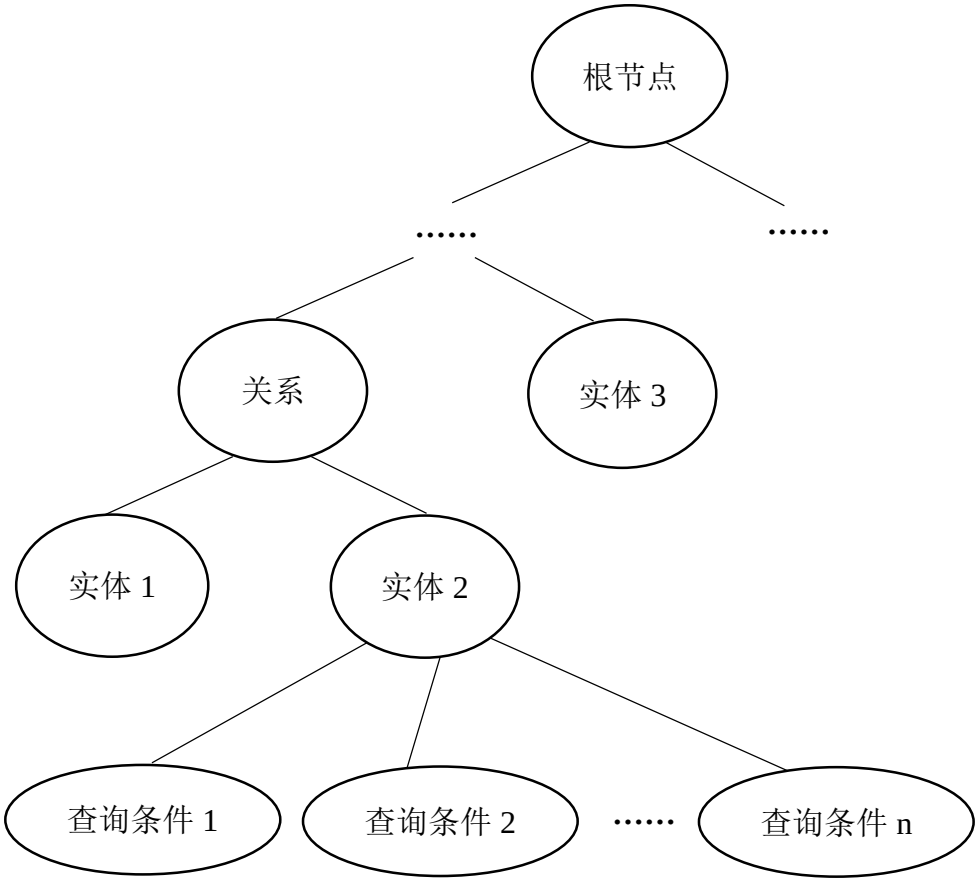


图 5-8 二叉查询树结构图

算法 5-2 二叉查询树构建算法

输入：实体关系集合，实体属性集合，返回条件
输出：构建完成的二叉查询树

```
joins = 从查询结构获取的实体关系集合;  //joins 表示实体关系集合
conditions = 从查询结构获取的实体属性集合;  //conditions 表示实体属性集合
returnObject = 从查询结构获取的返回条件;  //returnObject 表示返回条件
tree = new QueryTree();  // 查询树 tree 以 QueryTree 类进行初始化
If joins 为空
    返回;
Else joins 的大小=0 then
    //单个实体构建
```

```
9.      eNode = new EntityNode(conditions, returnObject);  //以实体属性集合、返回条件为参数构建实体节点 eNode
10.     tree.setRoot(Node);  //设置树的根节点
11.   Else
12.     //关系实体构建
13.     rootRDF = find()  //找到存有关系的 RDF 三元组
14.     joins.remove(rootRDF);  //在 joins 集合移除正在处理的三元组
15.     rNode = new RelationsNode(returnObject);  //构建关系节点 rNode
16.     rNode.setJoinColumn;  //设置关系列
17.     tree.setRoot(rNode);  //设置树的根节点
18.     rNode.setLeftChild(generateChild(rNode,rootRDF.left,joins,conditions));  // 以 generateChild 方法递归设置关系节点的左节点 Tag
19.     rNode.setRightChild(generateChild(rNode,rootRDF.right,joins,conditions));  //以 generateChild 方法递归设置关系节点的右节点 Tag
20.   End if
```

5.4.2 二叉查询树的遍历

在构建完二叉查询树后，系统就可以使用已实现的查询引擎（如采用内存迭代技术的查询引擎 MIQE，采用倒排索引技术的查询引擎 IIQE）自底向上以后序遍历的方式对二叉查询树节点遍历查询，获得查询结果。举例如下图 5-9 所示，有色框中的数字表示的是步骤，（1）表示的查询树是两个实体间的连接查询，（2）表示的查询树是三个实体间的连接实体查询。遍历算法如 5-3 所示。

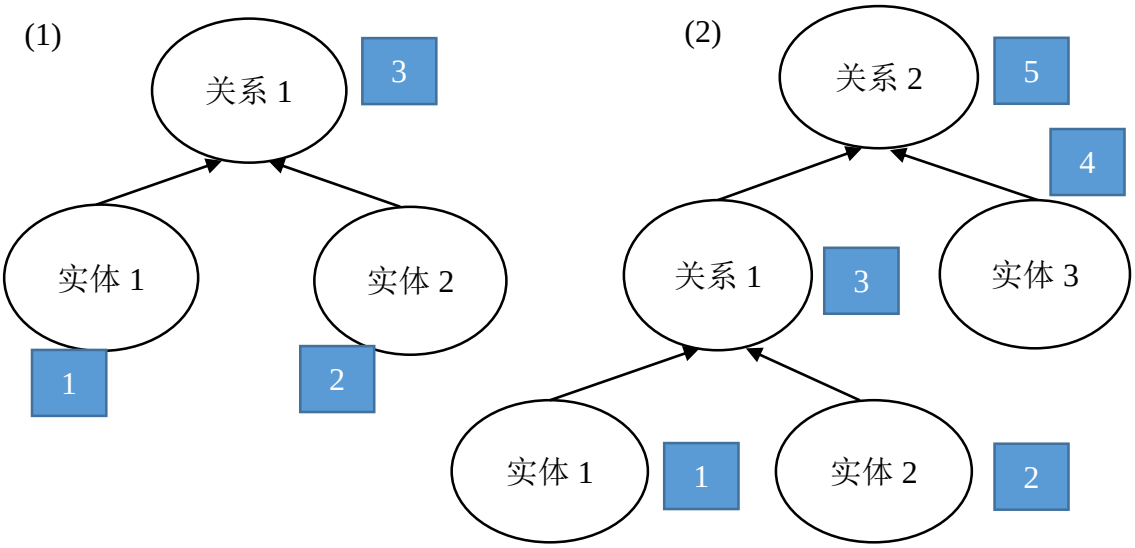


图 5-9 二叉查询树后序遍历

算法 5-3 二叉查询树遍历算法

输入： 二叉查询树
输出： 查询实体结果

```
1.    ResultSet postOrderRec(QueryNode root):   //ResultSet 表示结果对象，root 表示输入的根节点
2.        If(root != null)
3.            lchild = root 左节点的 Tag;
4.            rchild = root 右节点的 Tag;
5.            If(lchild != -1) then    //判断 lchild 是否存在
6.                左孩子节点  = postOrderRec(QueryNode.trees.get(lchild));   //从第 lchild 个节点获取左孩子节点
7.            End if
8.            If(rchild != -1) then    //判断 rchild 是否存在
9.                右孩子节点  = postOrderRec(QueryNode.trees.get(rchild));   //从第 rchild 个节点获取右孩子节点
10.           End if
11.           If(root 是关系节点)
12.               root 存储左右孩子节点;
13.           ResultSet results = root.query();   //通过 query 方法查询
14.           Return results;   //返回实体结果集
15.       End if
```

5.5 系统存储查询性能实验与分析

5.5.1 存储性能比较

本小节在 3.2 节的基础上对 Franke 存储模型，分布聚集存储模型和改进的分布聚集存储模型三种存储模型进行存储时间的比较，结果如下表 5-1 所示。从表 5-1 可以明显看出分布聚集存储模型的存储时间明显优于其余两种模型，改进的分布聚集存储模型由于要存储索引数据所以其存储时间明显大于只存储数据的分布聚集存储模型，Franke 存储模型由于要存储（SPO,OPS）两张表，所以存储时间较长但实验结果显示其明显大于其余两个存储模式一个数据量级。

表 5-1 三种存储模式存储时间结果

	Franke 存储模型	**分布聚集存储模型**	**改进的分布聚集存储模型**
一百万 RDF 数据集	132287049 ms	639836 ms	1482864 ms
一千万 RDF 数据集	1591316390 ms	6928636 ms	13704941 ms

5.5.2 查询性能比较

本节对上一章设计的采用内存迭代技术的查询引擎 MIQE 和采用倒排索引技术的查询引擎 IIQE 与 Franke 查询模型进行查询性能的比较。本节基于标准 LUBM 数据集分别在十万 RDF 数据集，百万 RDF 数据集和千万 RDF 数据集下进行单个实体查询和连接

查询的性能实验，其中单个实体查询使用常用的一个查询条件和两个查询条件的查询，连接查询使用常用的已知两个实体间关系和三个实体间关系的查询。由于知识图谱存储访问系统的数据载入模块和 SPARQL 语句解析模块均已在上文实现，因此针对三种模型的查询性能实验直接使用 SPARQL 语句进行查询。为了保证实验结果的准确性，每个实验本文都进行十次对应相同查询条件但查询内容不同的查询并在最后取平均查询时间，实验结果分别如下表 5-2，5-3 和表 5-4 所示。

表 5-2 十万 RDF 数据集的查询时间结果

	单个实体查询		联合实体查询	
	一个查询条件	两个查询条件	两个实体	三个实体
Franke	494 ms	405 ms	510 ms	1014 ms
MIQE	11505 ms	11669 ms	11820 ms	11900 ms
IIQE	432 ms	392 ms	520 ms	965 ms

表 5-3 一百万 RDF 数据集的查询时间结果

	单个实体查询		联合实体查询	
	一个查询条件	两个查询条件	两个实体	三个实体
Franke	886 ms	719 ms	856 ms	1910 ms
MIQE	30354 ms	33968 ms	33821 ms	41669 ms
IIQE	741 ms	642 ms	975 ms	1031 ms

表 5-4 一千万 RDF 数据集的查询时间结果

	单个实体查询		联合实体查询	
	一个查询条件	两个查询条件	两个实体	三个实体
Franke	2550 ms	687 ms	1820 ms	4558 ms
MIQE	80117 ms	80586 ms	85589 ms	84061 ms
IIQE	1003 ms	781 ms	1865 ms	2186 ms

从上三张表可以看出在各个 RDF 数据集下，各个查询模式的 MIQE 模型查询时间均明显大于 Franke 查询模型和 IIQE 模型查询时间。MIQE 模型在单个实体查询和连接查询下查询时间均为分钟级。经研究发现，造成 MIQE 模型查询时间较长的原因在于 Spark 初始化时间，由于本文 MIQE 需预先加载 HBase 中的所有数据，而加载数据的时间计算在实际查询时间中，所以 MIQE 的查询时间较长并且随着数据量的增大，由于 MIQE 要加载更多的数据，所以查询的时间也会增大。除去加载数据的时间，实验测得，在十万 RDF 数据集下，MIQE 模型的单个实体查询不超过 1 秒，连接查询不超过 2 秒，在一百万 RDF 数据集下，MIQE 模型单个实体查询不超过 3 秒，连接查询不超过 5 秒，在一千万 RDF 数据集下 MIQE 模型单个实体查询不超过 5 秒，连接查询不超过 10 秒，性能弱于 Franke 查询模型和 IIQE 模型。IIQE 模型的查询性能总体上优于 Franke 查询模型和 MIQE 模型，IIQE 模型在一个查询条件和两个查询条件的单个实体查询下的查询时间均不超过 2 秒，两个实体和三个实体的连接查询的查询时间均不超过 3 秒。Franke 模型的查询性能优于 MIQE 模型的查询性能，但是与 IIQE 模型相比虽然在两个查询条件下的单个实体查询和两个实体间的连接实体查询性能接近，但是一个查询条件下的单个实体查询，IIQE 查询性能明显优于 Franke 查询模型，三个实体间的连接查询 IIQE 模

型的查询性能更是比 Franke 查询模型快了 1 倍左右。在单个实体查询中，Franke 查询模型和 IIQE 模型在两个查询条件下的查询时间均快于一个查询条件下的查询时间，其主要原因是更多的查询条件会限制查询结果的数量，只有一个查询条件会由于结果数量多会造成数据多次的网络传输增加查询时间。综上所述，基于分布聚集存储模式和分布并行查询引擎实现的知识图谱存储访问系统具有良好的水平扩展性，IIQE 模型相比于其他两种查询模型更适合大规模知识图谱的查询。

5.6 系统展示

基于前几节讨论的知识图谱数据载入模块，SPARQL 语句解析模块和数据查询模块，本文构建了完整的基于大数据平台的知识图谱存储访问原型系统。系统的主要功能就是对知识图谱的实体和关系进行存储查询。本文设计的知识图谱存储访问系统充分考虑了知识图谱的存储查询的水平扩展性。

5.6.1 存储展示

下图 5-10 展示了部分存储在 HBase 中的知识图谱实体和索引数据。在此展示的原型系统中，系统对所有的宾语建立了索引。以第一行为例，第一行只显示了实体“<http://www.Department8.University10.edu/UndergraduateStudent104>”的索引，根据前缀可以发现，该实体存储在 Region“0000-0999”的 Region 块中，因此其索引行键的前缀为“0000”，该索引是基于“<D:\x5Cworkspace\x5CLUBM\x5Contology.owl#telephone>”谓语下的宾语“xxx-xxx-xxxx”建立的。

```
Current count: 308000, row: 0000:"xxx-xxx-xxxx":attributes:<D:\x5Cworkspace\x5CL
UBM\x5Contology.owl#telephone>_0665:<http://www.Department8.University10.edu/Und
ergraduateStudent104>
Current count: 309000, row: 0000:"xxx-xxx-xxxx":attributes:<D:\x5Cworkspace\x5CL
UBM\x5Contology.owl#telephone>_0677:<http://www.Department7.University39.edu/Und
ergraduateStudent296>
Current count: 310000, row: 0000:"xxx-xxx-xxxx":attributes:<D:\x5Cworkspace\x5CL
UBM\x5Contology.owl#telephone>_0689:<http://www.Department8.University17.edu/Und
ergraduateStudent48>
Current count: 311000, row: 0000:"xxx-xxx-xxxx":attributes:<D:\x5Cworkspace\x5CL
UBM\x5Contology.owl#telephone>_0701:<http://www.Department3.University90.edu/Und
ergraduateStudent339>
Current count: 312000, row: 0000:"xxx-xxx-xxxx":attributes:<D:\x5Cworkspace\x5CL
UBM\x5Contology.owl#telephone>_0713:<http://www.Department4.University55.edu/Und
ergraduateStudent139>
Current count: 313000, row: 0000:"xxx-xxx-xxxx":attributes:<D:\x5Cworkspace\x5CL
UBM\x5Contology.owl#telephone>_0725:<http://www.Department9.University18.edu/Gra
duateStudent55>
```

图 5-10 HBase 中的知识图谱实体和索引数据

5.6.2 查询展示

本节展示在一千万 RDF 数据集下一个查询条件的单个实体查询结果，两个查询条件的单个实体查询结果，两个实体间连接查询的查询结果，三个实体间连接查询的查询结果和两个实体的实体关系查询。

（1）一个查询条件的单个实体查询如下图 5-11 所示。该 SPARQL 查询语句旨在查询“emailAddress”为“GraduateStudent89@Department5.University7.edu”的所有实体。

查询语句返回该实体的所有信息，包括“emailAddress”，“memberof”和“name”等等所有属性。该 SPARQL 查询语句查询时间为 987ms。

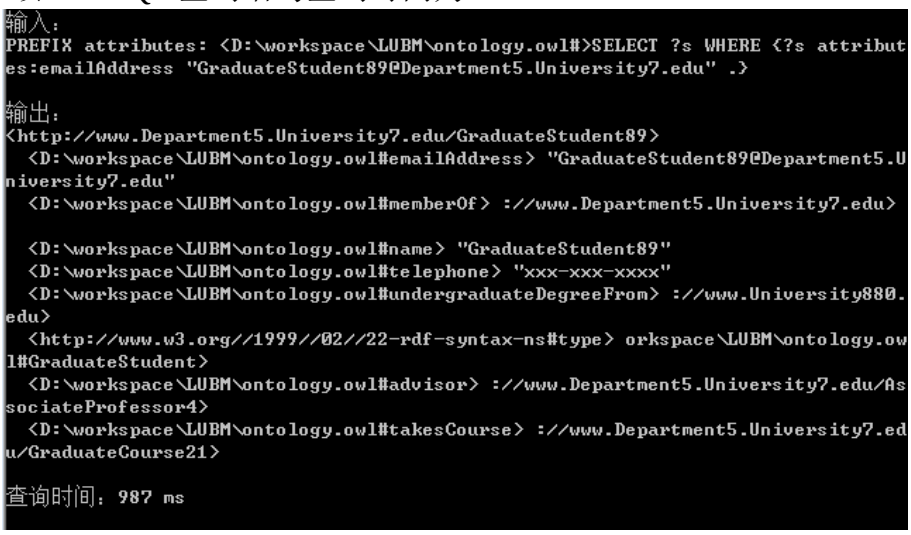


图 5-11 一个查询条件的单个实体查询

（2）两个查询条件的单个实体查询如下图 5-12 所示。该 SPARQL 查询语句旨在查询“name”为“GraduateStudent89”且“emailAddress”为“GraduateStudent89@Department5.University7.edu”的所有实体。查询语句返回符合查询条件的实体的所有信息，包括“emailAddress”，“memberof”和“name”等等所有属性。该 SPARQL 查询语句查询时间为 966ms。

图 5-12 两个查询条件的单个实体查询

（3）两个实体间的连接查询如下图 5-13 所示。该 SPARQL 查询语句旨在查询实体 1，其中已知实体 1 的“name”为“GraduateStudent89”，实体 1 与实体 2 存在“advisor”的关系，实体 2 的“name”为“AssociateProfessor4”。查询语句返回该实体 1 的所有信息，包括“emailAddress”，“memberof”和“name”等等所有属性。该 SPARQL 查询语句查询时间为 1907ms。

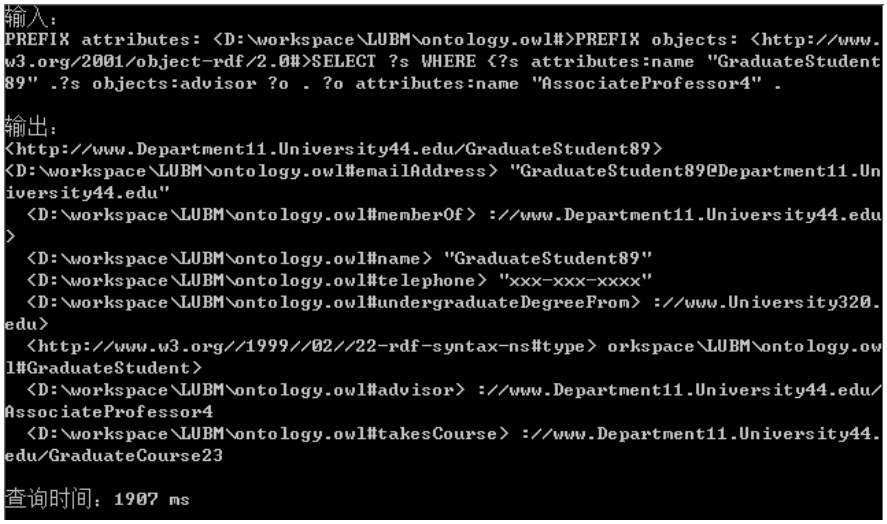


图 5-13 两个实体间的连接查询

（4）三个实体间的连接查询如下图 5-14 所示。该 SPARQL 查询语句旨在查询实体 1，其中已知实体 1 的“name”为“GraduateStudent89”，实体 1 与实体 2 存在“advisor”的关系，实体 2 的“name”为“AssociateProfessor4”，实体 2 与实体 3 存在“teacherOf”的关系，实体 3 的“name”为“GraduateCourse22”。查询语句返回该实体 1 的所有信息，包括“emailAddress”，“memberof”和“name”等等所有属性。该 SPARQL 查询语句查询时间为 2079ms。

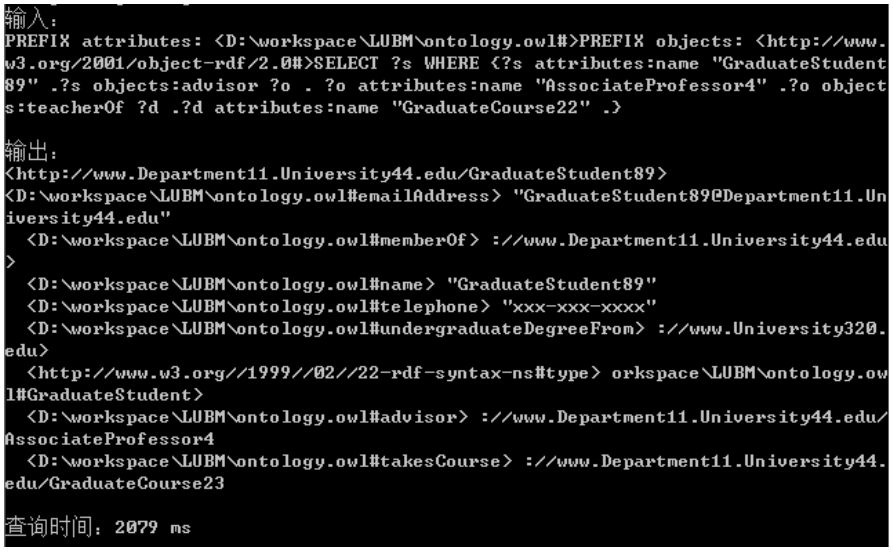


图 5-14 三个实体间的连接查询

（5）两个实体的实体关系查询如下图 5-15 所示。该 SPARQL 查询语句旨在查询两个实体间的未知关系，其中已知实体 1 的“name”为“GraduateStudent89”，实体 2 的“name”为“AssociateProfessor4”。查询语句返回实体 1 与实体 2 之间的关系“advisor”。由于实体间未知关系的查询较耗费时间，本系统不统计对实体关系查询的查询时间。

图 5-15 两个实体的实体关系查询

5.7 本章小结

本章首先描述了本文设计并实现的知识图谱存储访问原型系统的系统结构，分别介绍了系统的三大模块：数据载入模块，SPARQL 语句解析模块和数据查询模块。然后本章对系统进行了存储查询的性能测试，实验证明，IIQE 查询引擎的查询效率比 MIQE 查询引擎的查询效率要高，IIQE 查询引擎充分满足知识图谱查询可扩展的要求。最后，本章对已实现的基于大数据平台的知识图谱存储访问系统进行了系统展示。

第六章 总结与展望

6.1 总结

面对越来越多的数据，知识图谱存储访问系统迫切地需要可扩展的存储模式和分布并行的查询以保证知识图谱存储和查询的效率。本文设计出具有存储负载分布均衡、局部节点聚集存储的分布聚集存储模式和可以并行查询查询树的分布并行查询引擎以保证知识图谱的存储扩展性和查询扩展性。基于设计的分布聚集存储模式和分布并行查询引擎本文实现了基于大数据平台的知识图谱存储访问系统。下面是本文的主要工作：

（1）分布聚集存储模式的设计。基于 Big Table 模型稀疏、分布式、一致、多维排序的特性，本文设计出单表多列簇的分布聚集存储模式。该存储模式有着总体负载均衡，局部聚集的特点，满足知识图谱存储可扩展的要求。

（2）分布并行查询引擎的设计。为了加快查询速度，基于分布聚集存储模式设计出两种采用不同方案的分布并行查询引擎：MIQE 和 IIQE。MIQE 采用分布式内存迭代技术以过滤和连接操作并行查询抽象集合表示的实体，IIQE 采用倒排索引技术并利用集群分布并行扫描能力以归并的方式查询实体，两种分布并行查询引擎均满足知识图谱查询可扩展的要求。

（3）原型系统的实现和性能验证。基于上述研究，本文设计并实现基于大数据平台的知识图谱存储访问系统。系统分为三大模块：数据载入模块，SPARQL 语句解析模块和数据查询模块。知识图谱先通过数据载入模块载入到 HBase 中。查询时，SPARQL 查询语句通过解析模块获得查询结构，然后在数据查询模块生成查询树再进行基于 HBase 查询方式的查询。实验验证，IIQE 查询引擎查询性能比 MIQE 查询引擎性能优异。本文实现的知识图谱存储访问系统不仅可以高效地存储知识图谱而且保证知识图谱的查询性能。

6.2 未来工作

由于作者的水平和时间有限，对于知识图谱存储查询依然有进一步改进的地方，主要包括以下几个方面：

（1）完善丰富基于查询引擎 IIQE 的知识图谱存储查询。目前只是实现了常用的 SPARQL 语句查询 HBase 中存储的实体，对于复杂的 SPARQL 语句查询目前并不支持，需要进一步完善。

（2）优化查询树的生成，目前查询树是根据三元组依次出现的顺序进行构建，该方案可能会在面对大数据量的返回实体上造成查询性能的瓶颈，需要做类似于 SQL 语句谓词下推等方面的优化工作。

（3）升级查询方案，为了满足未来更多的查询要求和查询性能，可以考虑全文搜索引擎 ElasticSearch[42]与 HBase 相结合，将 ElasticSearch 优秀的检索性能结合到 HBase 中，提升查询性能。

参考文献

[1] Singhal A. Introducing the knowledge graph: things, not strings[J]. Official google blog, 2012.
[2] 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.
[3] Wilkinson K. Jena property table implementation[J]. Ssws, 2006.
[4] Carrol J, McBride B. The Jena Semantic Web Toolkit[J]. Public api, HP-Labs, Bristol, 2001.
[5] Abadi D J, Marcus A, Madden S R, et al. SW-Store: a vertically partitioned DBMS for Semantic Web data management[J]. Vldb Journal, 2009, 18(2):385-406.
[6] Franke C, Morin S, Chebotko A, et al. Efficient Processing of Semantic Web Queries in HBase and MySQL Cluster[J]. It Professional, 2013, 15(3):36-43.
[7] Ronstrom M, Thalmann L. MySQL cluster architecture overview[J]. MySQL Technical White Paper, 2004.
[8] http://swat.cse.lehigh.edu/projects/lubm/
[9] 卢传耀. 基于 NoSQL 的 RDF 存储与冗余消除的研究与实现[D]. 南京航空航天大学, 2014.
[10] Jain V, Upadhyay A. MongoDB and NoSQL Databases[J]. International Journal of Computer Applications, 2017, 167(10).
[11] Chodorow K, Dirolf M. MongoDB: The Definitive Guide[M]. 东南大学出版社, 2014.
[12] 辛晓越. 文档型数据库的存储模型设计和研究[D]. 中山大学, 2015.
[13] Mcbride B. Jena: A Semantic Web Toolkit[M]. IEEE Educational Activities Department, 2002.
[14] George L. HBase : the definitive guide[J]. Andre, 2011, 12(1):1 - 4.
[15] Kim D J, Shin J H, Hong K S. Scalable RDF store based on HBase and MapReduce[C]// International Conference on Advanced Computer Theory and Engineering. IEEE, 2010:V1-633 - V1-636.
[16] White T. Hadoop: The definitive guide[M]. " O'Reilly Media, Inc.", 2012.
[17] Li K, Wu B, Wang B. A Distributed RDF Storage and Query Model Based on HBase[C]// International Conference on Web-Age Information Management. Springer, Cham, 2015:3-15.
[18] 王林彬, 黎建辉, 沈志宏. 基于 NoSQL 的 RDF 数据存储与查询技术综述[J]. 计算机应用研究, 2015(5):1281-1286.
[19] 王仁武, 袁毅, 袁旭萍. 基于深度学习与图数据库构建中文商业知识图谱的探索研究[J]. 图书与情报, 2016(1):110-117.
[20] Van Bruggen R. Learning Neo4j[M]. Packt Publishing Ltd, 2014.
[21] Berners-Lee T, Hendler J, Lassila O. The semantic web[J]. Scientific american, 2001, 284(5): 28-37.
[22] Allemang D, Hendler J. Semantic Web for the Working Ontologist: Effective Modeling in RDFSand OWL[J]. Semantic Web for the Working Ontologist, 2008, 14(3):343–346.
[23] Dan B, Guha R V. RDF Vocabulary description language 1.0: RDF, schema[J]. Department of Computer Science University of Helsinki, 2004.
[24] 希茨利尔, P.). 语义 Web 技术基础[M]. 清华大学出版社, 2012.
[25] 王秋月, 覃雄派, 曹巍,等. 扩展知识图谱上的实体关系检索[J]. 计算机应用, 2016, 36(4):985-991.
[26] World Wide Web Consortium. RDF 1.1 concepts and abstract syntax[J]. 2014.

[27] Lehmann J. DBpedia: A large-scale, multilingual knowledge base extracted from wikipedia[J]. Semantic Web, 2015, 6(2):167-195.
[28] Prud'hommeaux, Eric, Seaborne, et al. SPARQL Query Language for RDF[J]. 2008, 4.
[29] ArangoDB, https://www.arangodb.org
[30] Lamport L, Reed B C, Junqueira F P, et al. In Search of an Understandable Consensus Algorithm[J]. Draft of October, 2014.
[31] http://cryto.net/~joepie91/blog/2015/07/19/why-you-should-never-ever-ever-use-mongodb/
[32] Panzarino O. Learning Cypher[M]. Packt Publishing Ltd, 2014.
[33] http://spark.apache.org/
[34] 陈欢, 林世飞. Spark 最佳实践[M]. 人民邮电出版社, 2016.
[35] Wu X, Xu Y, Shao Z, et al. LSM-trie: an LSM-tree-based ultra-large key-value store for small data[C]//Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference. USENIX Association, 2015: 71-82.
[36] http://swat.cse.lehigh.edu/projects/lubm/
[37] Nayak A, Poriya A, Poojary D. Type of NOSQL databases and its comparison with relational databases[J]. International Journal of Applied Information Systems, 2013, 5(4): 16-19.
[38] Wang Y, Li C, Li M, et al. HBase storage schemas for massive spatial vector data[J]. Cluster Computing, 2017(2):1-10.
[39] George L. HBase Coprocessors - Deploy shared functionality directly on the cluster - O'Reilly Media Free, Live Events[J]. Journal of East China University Ofence & Technology, 2017.
[40] Team A H B. Apache hbase reference guide[J]. Apache, version, 2016, 2(0).
[41] Parr T. The definitive ANTLR 4 reference[M]. Pragmatic Bookshelf, 2013.
[42] Gormley C, Tong Z. Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine[M]. " O'Reilly Media, Inc.", 2015.